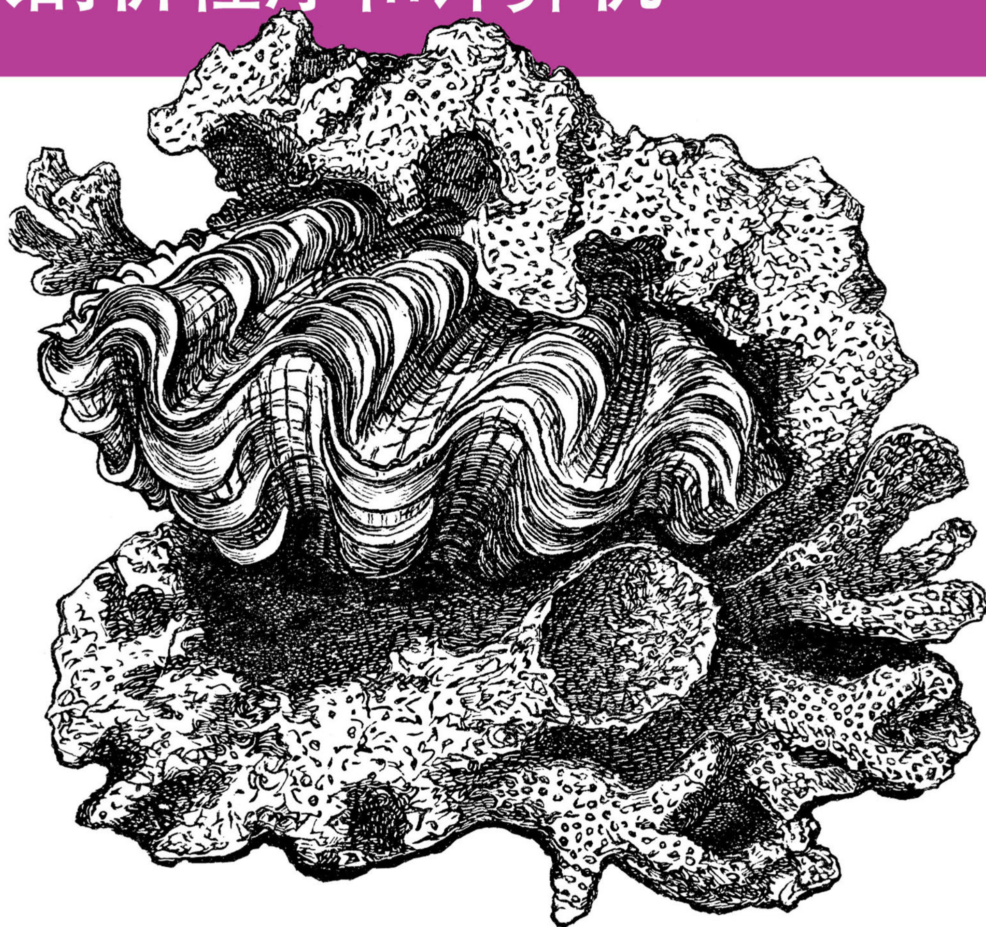
TURING 图灵程序设计丛书

Understanding Computation: From Simple Machines to Impossible Programs

计算的本质

深入剖析程序和计算机



[英] *Tom Stuart* 著

张伟 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

亚马逊读者评论

“Tom讲解关键概念的方式出人意料，详略得当，一语中的。当年我上大学时经常纠结于这些概念……当初若有这本书，该有多好啊！”

“虽然本书讲述了深刻的计算理论，但我却觉得这是一次愉悦的阅读之旅。书中短小精悍的Ruby示例让我惊喜连连。本书确实技术性很强，但恰到好处的示例让我们很容易理解各种概念，其系统的梳理和讲解绝对会让你不虚此行。”

“我刚刚读完前三章的内容，就已了解如何用一种编程语言实现另一种编程语言了！当然，我还学到了一些Ruby知识。本书内容详略得当、易于理解。Tom把原本枯燥的主题讲解得妙趣横生，同时又极具启发性。”


图灵程序设计丛书

计算的本质
深入剖析程序和计算机

Understanding Computation
From Simple Machines to Impossible Programs

[英] Tom Stuart 著
张伟 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

计算的本质 : 深入剖析程序和计算机 / (英) 斯图尔特 (Stuart,T.) 著 ; 张伟译. -- 北京 : 人民邮电出版社, 2014.11（2016.6重印）
（图灵程序设计丛书）
ISBN 978-7-115-36154-7

Ⅰ. ①计… Ⅱ. ①斯… ②张… Ⅲ. ①程序设计－基本知识 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2014)第135541号

内 容 提 要

本书借助简单的 Ruby 代码示例，全面、深入地介绍计算理论和编程语言设计。作者注重实用性，在读者熟知的背景知识下，以明晰的可工作代码阐释了形式语义、自动机理论，以及通过 lambda 演算进行函数式编程等计算问题，并为读者自行探索打下了良好基础。

本书面向熟悉某种现代编程语言却非科班出身的程序员，是一本帮你真正理解计算机科学和计算原理的优秀参考书。

◆ 著　　　　[英] Tom Stuart
译　　　　张　伟
责任编辑　李松峰　毛倩倩
执行编辑　程　芃
责任印制　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：18.75
字数：433千字　　　　2014年 11 月第 1 版
印数：3 701 – 4 000册　　　　2016年 6 月北京第 3 次印刷

著作权合同登记号　图字：01-2013-5148号

定价：69.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

第一部分 程序和机器

第二部分　计算与可计算性

封面介绍

本书封面上的动物是砗蚝（拉丁学名为 Hippopus hippopus）。砗蚝因其形状又叫马蹄蛤，又因其颜色泛红称为草莓蛤。砗蚝是砗磲科巨蛤亚科的一部分，而砗磲科又是乌蛤科的一部分。砗蚝主要生活在印度洋－太平洋区域的礁石中。

砗蚝有两个相同而对称的铰合部。它还有着深深的嵴和与众不同的红白图案。砗蚝待在一个地方使用虹管过滤周围的水之后，以周围的浮游生物为食。

前言

读者对象

本书写给那些对编程语言和计算理论充满好奇的程序员，特别是没有正规学习过数学或者计算机科学的朋友。

如果你对涉及程序、语言以及机器，且能开阔思维的计算机科学知识感兴趣，却被常常用于阐明它们的数学语言打击的话，那么本书恰恰是你需要的。我们抛开复杂的数学符号，用可工作的代码来描述理论性概念，并为大家自行探索做足准备。

本书读者至少要了解一种现代编程语言，如 Ruby、Python、JavaScript、Java 或者 C#。书中所有示例程序都采用 Ruby 语言编写而成，但了解其他语言的读者亦能看懂。注意，本书目标并不是展示 Ruby 或面向对象设计的最佳实践。本书代码意在简明清晰，但并不一定都容易维护，因为我们的目标是使用 Ruby 阐明计算机科学，而不是用计算机科学讲解 Ruby。本书亦非教材或者百科全书，所以并没有给出形式论证或者严密的证明，它试图让你能接近一些有趣的思想，启发你更深入地了解它们。

排版约定

本书中使用以下排版约定。

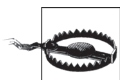
- 楷体
  用于标记新名词。

- 等宽字体（`constant width`）
  用于程序代码，在段落中用于表示程序的组成部分，如变量或函数名、数据库、数据类型、环境变量、语句、关键字。

- 等宽粗体（**`constant width bold`**）
  命令或是其他应该由用户输入的内容。
- 等宽斜体（*`constant width italic`*）
  应该由用户提供或由上下文确定的值。



提示、建议或一般注解会放在这里。



警告或警示信息会放在这里。

使用代码

本书旨在帮助读者解决实际问题。也许你需要在自己的程序或文档中用到本书中的代码，但除非大段大段地使用，否则不必与我们联系取得授权。因此，用本书中的几段代码写个程序不用向我们申请许可，但是销售或者分发 O'Reilly 图书随附的代码光盘则必须事先获得授权；引用书中的代码来回答问题也无需我们授权，而将大段的示例代码整合到自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版商和书号，例如："*Understanding Computation by Tom Stuart* (O'Reilly). Copyright 2013 Tom Stuart, 978-1-4493-2927-3"。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com。

Safari® Books Online



Safari Books Online（www.safaribooksonline.com）是应需而变的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online 是技术专家、软件开发人员、Web 设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM

Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://oreil.ly/Understanding_Computation

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

非常感谢 Go Free Range 的热情款待，他们在本书写作期间为我提供了办公地点、友好的交流以及茶水。没有他们慷慨的支持，我肯定早就放弃了。

感谢 James Adam、Paul Battley、James Coglan、Peter Fletcher、Chris Lowis 和 Murray Steele 对早期草稿的反馈，并且感谢 Gabriel Kerneis 和 Alex Stangl 的技术审校。有了他们的贡献，本书的质量大为提高。我还要对剑桥大学的 Alan Mycroft 提供资料和支持表示感谢。

O'Reilly 的许多人帮助保证了本书的顺利完工与出版，我要特别感谢 Mike Loukides 和 Simon St.Laurent 一开始就对这个想法抱有热情和信念，感谢 Nathan Jepson 对如何把想法转变成实际图书的建议，以及 Sanders Kleinfeld 对我要求完美的语法高亮这一严苛要求的迁就。

感谢我的父母让一个恼人的小孩有办法、有动机，也有机会在计算机上混日子。还要感谢 Leila，每次我忘记工作应该如何完成时，她都会耐心地提醒我把该死的词儿一个接一个地排起来。最后我做到了。

第 1 章

刚好够用的Ruby基础

本书中的代码全部使用 Ruby 写成。Ruby 是一种简单、友好而且有趣的编程语言。因为 Ruby 清晰与灵活，我选择了它，但本书并不依赖于 Ruby 专有的特性，所以这些示例代码均可转换成你喜欢的其他任何语言，特别是像 Python 或者 JavaScript 这样的动态语言，如果那样你更容易理解的话。

所有的示例代码都兼容 Ruby 2.0 和 Ruby 1.9。你可以在 Ruby 官方站点（http://www.ruby-lang.org/）详细了解 Ruby，还可以下载一份官方的实现。

我们会快速浏览一下 Ruby 的特性，并集中介绍本书中用到的部分。如果你想学习更多内容，推荐从 O'Reilly 的《Ruby 编程语言》（*The Ruby Programming Language*）一书起步。



如果已经了解 Ruby，你完全可以从第 2 章开始阅读本书。

1.1　交互式Ruby Shell

Ruby 最友好的一个特性就是交互式控制台 IRB，它可以让我们在输入 Ruby 代码后立即看到执行结果。本书将广泛使用 IRB 与所写的代码进行交互，并探索这些代码是如何工作的。

在开发机器的命令行中输入 `irb`，就可以运行 IRB 了。IRB 显示提示符 >> 时，表明当前可以输入一个 Ruby 表达式。输入一个表达式并敲回车键之后，代码执行，结果会显示到提示符 => 之后：

```
$ irb --simple-prompt
>> 1 + 2
=> 3
>> 'hello world'.length
=> 11
```

本书中只要出现提示符 >> 和 =>，就是在与 IRB 交互。为了让长代码更易读，本书显示它们的时候会去掉提示符，但是仍然假定这些代码已经输入或者粘贴进了 IRB。所以一旦本书中有像下面这样的 Ruby 代码：

```
x = 2
y = 3
z = x + y
```

我们之后就可以在 IRB 中得到它们的结果：

```
>> x * y * z
=> 30
```

1.2 值

Ruby 是一种*面向表达式*的语言：每一段有效的代码执行之后都要产生一个值。下面快速浏览一下 Ruby 中不同类型的值。

1.2.1 基本数据

如我们所料，Ruby 支持布尔型（Boolean）、数值型（number）和字符串（string），且它们都支持常规运算：

```
>> (true && false) || true
=> true
>> (3 + 3) * (14 / 2)
=> 42
>> 'hello' + ' world'
=> "hello world"
>> 'hello world'.slice(6)
=> "w"
```

一个 Ruby *符号*表示一个名字，是一个轻量级、不可变的值。作为字符串的简单化、非内存密集化（less memory-intensive）的替身，符号在 Ruby 中被广泛使用——通常是作为散列表中的键使用（参见 1.2.2 节）。符号字面量的开头会有一个冒号：

```
>> :my_symbol
=> :my_symbol
>> :my_symbol == :my_symbol
=> true
>> :my_symbol == :another_symbol
=> false
```

特殊值 nil 用来表示不存在任何有用的值：

```
>> 'hello world'.slice(11)
=> nil
```

1.2.2 数据结构

Ruby 的数组字面量是一串用逗号分隔的值外加方括号的形式：

```
>> numbers = ['zero', 'one', 'two']
=> ["zero", "one", "two"]
>> numbers[1]
=> "one"
>> numbers.push('three', 'four')
=> ["zero", "one", "two", "three", "four"]
>> numbers
=> ["zero", "one", "two", "three", "four"]
>> numbers.drop(2)
=> ["two", "three", "four"]
```

范围（range）表示最小值和最大值之间值的集合。范围的写法是在两个值之间加两个点：

```
>> ages = 18..30
=> 18..30
>> ages.entries
=> [18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
>> ages.include?(25)
=> true
>> ages.include?(33)
=> false
```

一个散列（hash）表示一个集合，其中每个值都与一个键相关联；一些编程语言把这种数据结构叫作“映射”（map）、“字典”（dictionary）或者“关联数组”（associative array）。一个散列字面量写成大括号里用逗号分隔的“键 => 值”对的列表：

```
>> fruit = { 'a' => 'apple', 'b' => 'banana', 'c' => 'coconut' }
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut"}
>> fruit['b']
=> "banana"
>> fruit['d'] = 'date'
=> "date"
>> fruit
=> {"a"=>"apple", "b"=>"banana", "c"=>"coconut", "d"=>"date"}
```

散列经常将符号用作键，所以键作为符号时，Ruby 提供了另一种书写键值对的语法。这种写法比“键 => 值”的方式更为紧凑，而且看起来很像常用于 JavaScript 对象的 JSON 格式：

```
>> dimensions = { width: 1000, height: 2250, depth: 250 }
=> {:width=>1000, :height=>2250, :depth=>250}
>> dimensions[:depth]
=> 250
```

1.2.3 proc

一个 proc 是一段未经求值的 Ruby 代码，根据需要进行传递和求值；其他语言把这种语言结构称为“匿名函数”或“lambda 函数”。proc 字面量有多种写法，其中最紧凑的一种是“-> 参数 { 函数体 }”语法：

```
>> multiply = -> x, y { x * y }
=> #<Proc (lambda)>
>> multiply.call(6, 9)
=> 54
>> multiply.call(2, 3)
=> 6
```

除了 `.call` 语法，还可以使用方括号调用 proc:

```
>> multiply[3, 4]
=> 12
```

1.3 控制流

Ruby 有 `if`、`case` 和 `while` 表达式，它们都以通常的方式工作：

```
>> if 2 < 3
     'less'
   else
     'more'
   end
=> "less"
>> quantify =
     -> number {
       case number
       when 1
         'one'
       when 2
         'a couple'
       else
         'many'
       end
     }
=> #<Proc (lambda)>
>> quantify.call(2)
=> "a couple"
>> quantify.call(10)
=> "many"
>> x = 1
=> 1
>> while x < 1000
     x = x * 2
   end
=> nil
>> x
=> 1024
```

1.4 对象和方法

Ruby 看起来和其他动态编程语言很像，但有一个重要的区别：每个值都是一个对象，而且对象彼此之间靠发送消息进行通信[1]。每个对象都有自己的方法集合，这些方法决定了它如何响应特定的消息。

一个消息有一个名字，并且根据需要可以有一些参数。一个对象收到一个消息的时候，它对应的方法就会使用消息中的参数作为自己的参数执行。这就是 Ruby 完成全部工作的方式；甚至“1+2”都意味着“使用参数 2 给对象 1 发送一个叫作 + 的消息”，而对象 1 有一个处理那个消息的方法 #+。

我们可以使用关键字 def 定义自己的方法：

```
>> o = Object.new
=> #<Object>
>> def o.add(x, y)
     x + y
   end
=> nil
>> o.add(2, 3)
=> 5
```

这里，我们通过向一个特殊内建对象 Object 发送 new 消息来新建一个对象；新对象创建之后，在其上定义了一个叫 #add 的方法。#add 方法把它的两个参数加在一起，并返回结果；因为一个方法中最后执行的表达式的值将被自动返回，所以并不需要一个显式的 return。在使用 2 和 3 作为参数向那个对象发送 add 消息之后，#add 方法就会执行，然后我们就得到了想要的结果。

通常情况下，在发送消息时要写上接收对象和消息名并用圆点分隔（例如 o.add），但是 Ruby 会一直追踪当前对象（叫作 self），这样在向当前对象发送消息时只需写上一个消息名，接收对象可以不必显式写出来。例如，在一个方法定义内部，当前对象总是接收消息并执行此方法的对象，因此在一个特定对象的方法内部，向同一个对象发送其他消息时，可以不必显式提及：

```
>> def o.add_twice(x, y)
     add(x, y) + add(x, y)
   end
=> nil
>> o.add_twice(2, 3)
=> 10
```

注意，我们在 #add_twice 方法里给 o 发送 add 消息时，可以不必写成 o.add(x, y)，只写

注 1：这种来自于编程语言 Smalltalk 的风格，对 Ruby 的设计有直接影响。

add(x,y) 就可以，这是因为 o 是接收 add_twice 消息的对象。

在所有的方法定义之外，当前对象是一个叫 main 的特殊顶层对象，任何没有指明接收者的消息都会被发送给它；同样，任何没有指明对象的方法定义都可以通过 main 使用：

```
>> def multiply(a, b)
     a * b
   end
=> nil
>> multiply(2, 3)
=> 6
```

1.5 类和模块

能在许多对象之间共享方法定义是件很便利的事。在 Ruby 中我们可以把方法定义放到一个类里，然后通过给那个类发送 new 消息来新建对象。所获得的对象是包括方法在内的这个类的实例。例如：

```
>> class Calculator
     def divide(x, y)
       x / y
     end
   end
=> nil
>> c = Calculator.new
=> #<Calculator>
>> c.class
=> Calculator
>> c.divide(10, 2)
=> 5
```

注意，在一个类定义里定义一个方法会把方法添加到那个类的实例里，而不是加到 main 里：

```
>> divide(10, 2)
NoMethodError: undefined method `divide' for main:Object
```

一个类可以通过继承来引入另一个类的方法定义：

```
>> class MultiplyingCalculator < Calculator
     def multiply(x, y)
       x * y
     end
   end
=> nil
>> mc = MultiplyingCalculator.new
=> #<MultiplyingCalculator>
>> mc.class
=> MultiplyingCalculator
>> mc.class.superclass
=> Calculator
```

```
>> mc.multiply(10, 2)
=> 20
>> mc.divide(10, 2)
=> 5
```

子类中的方法可以通过 super 关键字调用超类的同名方法：

```
>> class BinaryMultiplyingCalculator < MultiplyingCalculator
     def multiply(x, y)
       result = super(x, y)
       result.to_s(2)
     end
   end
=> nil
>> bmc = BinaryMultiplyingCalculator.new
=> #<BinaryMultiplyingCalculator>
>> bmc.multiply(10, 2)
=> "10100"
```

另一种共享方法定义的方式是在模块（module）中声明它们，这样它们就能被任意类包括进去：

```
>> module Addition
     def add(x, y)
       x + y
     end
   end
=> nil
>> class AddingCalculator
     include Addition
   end
=> AddingCalculator
>> ac = AddingCalculator.new
=> #<AddingCalculator>
>> ac.add(10, 2)
=> 12
```

1.6 其他特性

下面是本书中示例代码会用到的其他特性。

1.6.1 局部变量和赋值

就像我们已经看到的那样，Ruby 仅允许通过赋值声明局部变量：

```
>> greeting = 'hello'
=> "hello"
>> greeting
=> "hello"
```

我们还可以通过数组一次给多个变量并行赋值：

```
>> width, height, depth = [1000, 2250, 250]
=> [1000, 2250, 250]
>> height
=> 2250
```

1.6.2 字符串插值

字符串可以使用单引号也可以使用双引号表示。对双引号中的字符串，Ruby 会自动用表达式的结果替换 #{ 表达式 }，以执行字符串插值操作。

```
>> "hello #{'dlrow'.reverse}"
=> "hello world"
```

如果被插入的表达式返回的不是一个字符串类型的对象，那么这个对象就会自动收到一个 to_s 消息以返回能顶替其位置的字符串。我们可以借此控制被替换对象的展示方式：

```
>> o = Object.new
=> #<Object>
>> def o.to_s
     'a new object'
   end
=> nil
>> "here is #{o}"
=> "here is a new object"
```

1.6.3 检查对象

每当 IRB 需要显示一个对象，类似下面的一些事情就会发生：向这个对象发送 inspect 消息，然后这个对象返回自身的字符串表示。Ruby 当中所有对象默认都有对 #inspect 的合理实现，但是通过提供自己的定义，我们就可以控制如何在控制台显示对象：

```
>> o = Object.new
=> #<Object>
>> def o.inspect
     '[my object]'
   end
=> nil
>> o
=> [my object]
```

1.6.4 打印字符串

方法 #puts 对每个 Ruby 对象（包括 main）都可用，可以用来向标准输出打印字符串：

```
>> x = 128
=> 128
```

```
>> while x < 1000
     puts "x is #{x}"
     x = x * 2
   end
x is 128
x is 256
x is 512
=> nil
```

1.6.5 可变参数方法（variadic method）

定义方法时可以使用 * 运算符，以支持数目可变的参数：

```
>> def join_with_commas(*words)
     words.join(', ')
   end
=> nil
>> join_with_commas('one', 'two', 'three')
=> "one, two, three"
```

一个方法定义只能有一个可变参数，而常规参数放到可变参数的前后都可以：

```
>> def join_with_commas(before, *words, after)
     before + words.join(', ') + after
   end
=> nil
>> join_with_commas('Testing: ', 'one', 'two', 'three', '.')
=> "Testing: one, two, three."
```

在发送消息的时候，* 运算符还可以把每一个数组元素当作单个参数处理：

```
>> arguments = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> join_with_commas(*arguments)
=> "Testing: one, two, three."
```

* 也可以使用并行赋值方式：

```
>> before, *words, after = ['Testing: ', 'one', 'two', 'three', '.']
=> ["Testing: ", "one", "two", "three", "."]
>> before
=> "Testing: "
>> words
=> ["one", "two", "three"]
>> after
=> "."
```

1.6.6 代码块

代码块（block）是由 do/end 或者大括号围住的一段 Ruby 代码。方法可以带一个隐式代码块参数，并使用 yield 关键字表示对代码块中那段代码的调用：

```
>> def do_three_times
     yield
     yield
     yield
   end
=> nil
>> do_three_times { puts 'hello' }
hello
hello
hello
=> nil
```

代码块可以带参数：

```
>> def do_three_times
     yield('first')
     yield('second')
     yield('third')
   end
=> nil
>> do_three_times { |n| puts "#{n}: hello" }
first: hello
second: hello
third: hello
=> nil
```

yield 返回执行代码块的结果：

```
>> def number_names
     [yield('one'), yield('two'), yield('three')].join(', ')
   end
=> nil
>> number_names { |name| name.upcase.reverse }
=> "ENO, OWT, EERHT"
```

1.6.7 枚举类型

Ruby 有一个叫作 Enumerable 的内置模块，被数组（Array）、散列表（Hash）、范围（Range）以及其他表示值的集合的类包含。Enumerable 提供的方法可以帮助我们对集合进行遍历、搜索和排序，其中的很多方法在调用时都可以带上一个代码块。通常，代码块中的代码会根据集合中的一些值或全部值来运行，以此承担方法的一部分工作。例如：

```
>> (1..10).count { |number| number.even? }
=> 5
>> (1..10).select { |number| number.even? }
=> [2, 4, 6, 8, 10]
>> (1..10).any? { |number| number < 8 }
=> true
>> (1..10).all? { |number| number < 8 }
=> false
>> (1..5).each do |number|
```

```
    if number.even?
      puts "#{number} is even"
    else
      puts "#{number} is odd"
    end
  end
1 is odd
2 is even
3 is odd
4 is even
5 is odd
=> 1..5
>> (1..10).map { |number| number * 3 }
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

通常，一个代码块带有一个参数，并向此参数发送一个无参的消息，所以 Ruby 提供了一种缩写方式 `&:`*message*，这比写代码块 `{ |object| object.`*message* `}` 更为简洁：

```
>> (1..10).select(&:even?)
=> [2, 4, 6, 8, 10]
>> ['one', 'two', 'three'].map(&:upcase)
=> ["ONE", "TWO", "THREE"]
```

有的代码块可以为集合中的每个值生成一个数组，`Enumerable` 的方法 `#flat_map` 能把这些生成的结果数组连接起来：

```
>> ['one', 'two', 'three'].map(&:chars)
=> [["o", "n", "e"], ["t", "w", "o"], ["t", "h", "r", "e", "e"]]
>> ['one', 'two', 'three'].flat_map(&:chars)
=> ["o", "n", "e", "t", "w", "o", "t", "h", "r", "e", "e"]
```

还有一个有用的方法 `#inject`。有些代码块会处理集合中的每个值，`#inject` 能对这个代码块求值并累积成一个最终结果：

```
>> (1..10).inject(0) { |result, number| result + number }
=> 55
>> (1..10).inject(1) { |result, number| result * number }
=> 3628800
>> ['one', 'two', 'three'].inject('Words:') { |result, word| "#{result} #{word}" }
=> "Words: one two three"
```

1.6.8 结构体

结构体（`Struct`）是 Ruby 中一个特殊的类，它的工作是生成其他类。根据传进 `Struct.new` 的每个属性名，`Struct` 产生的类会包含相应的获取方法和设置方法。要使用由结构体生成的类，常见方式是对其进行子类化；我们可以给子类起个名字，然后在里边定义其他任意的方法。例如，为了创建一个拥有属性 `x` 和 `y`，名字是 `Point` 的类，可以写成：

```
class Point < Struct.new(:x, :y)
```

```
  def +(other_point)
    Point.new(x + other_point.x, y + other_point.y)
  end

  def inspect
      "#<Point (#{x}, #{y})>"
  end
end
```

现在我们可以创建 Point 的一些实例，然后在 IRB 中进行检查，并给它们发送消息：

```
>> a = Point.new(2, 3)
=> #<Point (2, 3)>
>> b = Point.new(10, 20)
=> #<Point (10, 20)>
>> a + b
=> #<Point (12, 23)>
```

和我们定义的所有方法一样，Point 实例会响应消息 x 和 x=，以便获取和设置属性 x 的值。y 和 y= 与 x 和 x= 的情况类似：

```
>> a.x
=> 2
>> a.x = 35
=> 35
>> a + b
=> #<Point (45, 23)>
```

由 Struct.new 生成的类还有其他实用功能，像判断是否相等的方法 #== 的实现，就可以比较两个结构体的属性是否相等：

```
>> Point.new(4, 5) == Point.new(4, 5)
=> true
>> Point.new(4, 5) == Point.new(6, 7)
=> false
```

1.6.9　给内置对象扩展方法（Monkey Patching）

我们随时都可以给类或模块增加方法。这是一个强大的特性，通常叫作 Monkey Patching，可以让我们扩展已有类的行为：

```
>> class Point
     def -(other_point)
       Point.new(x - other_point.x, y - other_point.y)
     end
   end
=> nil
>> Point.new(10, 15) - Point.new(1, 1)
=> #<Point (9, 14)>
```

我们甚至可以扩展 Ruby 内置的类：

```
>> class String
     def shout
       upcase + '!!!'
     end
   end
=> nil
>> 'hello world'.shout
=> "HELLO WORLD!!!"
```

1.6.10 定义常量

Ruby 支持一种叫作常量的特殊变量。一般而言，常量一旦创建，就不能再被重新赋值。（Ruby 并不会阻止一个常量被重新赋值，但它会产生警告，以便我们知道自己做错了事。）任何以大写字母开头的变量都是常量。可以在顶层或者在一个类或模块中定义新的常量：

```
>> NUMBERS = [4, 8, 15, 16, 23, 42]
=> [4, 8, 15, 16, 23, 42]
>> class Greetings
     ENGLISH = 'hello'
     FRENCH  = 'bonjour'
     GERMAN  = 'guten Tag'
   end
=> "guten Tag"
>> NUMBERS.last
=> 42
>> Greetings::FRENCH
=> "bonjour"
```

类和模块的名字总是以大写字母开头，所以类和模块的名字也是常量。

1.6.11 删除常量

在使用 IRB 进行探索时，如果我们想重新定义某个类或模块，而不是要扩展它们，实用的做法是让 Ruby 完全忽略该常量。一个顶层常量可以通过给 `Object` 发送消息 `remove_const` 来删除，同时还要把常量名作为符号（symbol）对象传进去：

```
>> NUMBERS.last
=> 42
>> Object.send(:remove_const, :NUMBERS)
=> [4, 8, 15, 16, 23, 42]
>> NUMBERS.last
NameError: uninitialized constant NUMBERS
>> Greetings::GERMAN
=> "guten Tag"
>> Object.send(:remove_const, :Greetings)
=> Greetings
>> Greetings::GERMAN
```

```
NameError: uninitialized constant Greetings
```

只能使用 Object.send(:remove_const，:常量名)而非 Object.remove_const(:常量名)，这是因为 remove_const 是一个私有（private）方法，只能通过从 Object 类的自身内部发送消息来调用；使用 Object.send 时，我们可以暂时跳过这个限制。

第一部分
程序和机器

什么是*计算*？这个词对于不同人来说意思不同，但是每个人都会赞同这样一种理解：在一台计算机读取程序、运行程序、读入一些输入，并且最后产生一些输出的时候，肯定发生了某种计算。因此我们可以这样认为：计算就是指*计算机所做的事情*。

为了创造一个环境让这种熟悉的计算发生，需要三个基本要素：

- 一台*机器*，能够执行计算；
- 一种*语言*，用来编写这台机器能够理解的指令；
- 一个*程序*，用这种语言编写，描述机器应该具体执行哪些计算。

这部分内容就是关于机器、语言和程序的：它们是什么，行为如何，我们如何对其建模并展开研究，以及如何利用它们完成实际工作。通过研究这三要素，我们将对计算的含义以及它是如何发生的有更好的理解。

在第 2 章，我们将设计和实现一种简单的编程语言，并用几种不同的方法来研究这种语言的含义。理解了一种语言的含义，就可以把一段没有生命的源代码和一个动态的、正在执行的进程联系起来。每一种方法都能带给我们一个把程序运行起来的特定策略，而我们最终将用几种不同的方式来实现同一语言。

我们会发现编程是一门把一个准确定义的结构组装起来的艺术，这个结构能拆卸、分析，并最终被一台机器解释执行从而完成一次计算。更重要的是，我们还会发现实现编程语言既简单又有趣：尽管语法分析、解释和编译看起来很吓人，但实际摆弄起来其实会感觉简单又愉快。

如果没有机器来运行，程序本身没有多大用处。所以在第 3 章里，我们会设计非常简单的机器，以便执行基本的、硬编码的任务。有了这个简单的基础，我们在第 4 章会向更复杂的机器努力前进，并在第 5 章介绍如何设计能被软件控制的通用计算装置。

到第二部分的时候，我们将了解拥有计算能力的机器的全景：一些机器拥有非常有限的能力，一些机器用处更大但仍然令人沮丧地有一些限制，最后还有一些机器是我们知道如何构建的最强大的机器。

第2章

程序的含义

> 不准想，快点！就像直觉地把手指向月亮。记住，反应慢了就只能看到手指，而绝不能看到月亮的光华了。
>
> ——电影《龙争虎斗》，李小龙

编程语言，以及我们用编程语言所写的程序，这些都是软件工程师工作的基础。我们用编程语言和程序阐明复杂的想法，并在彼此之间交流这些想法，当然最重要的是在计算机中实现这些想法。就像人类社会没有自然语言就难以运转一样，全球的程序员都依赖编程语言传递和实现自己的想法，每一个有成效的程序都是实现更高层思想的基础。

程序员是注重实际的生物。程序员经常通过阅读文档、学习教程、研究现有的程序以及修改自己的简单程序来学习新的编程语言，而不会过多地思考那些程序有什么含义。有时候，学习的过程就像试错：我们试图通过看例子和文档来理解一个语言片段，然后会努力用这种语言写点什么，之后所有问题就都爆发了，而我们只得回头重试，直到成功组装了一个大部分情况下都能工作的东西。随着程序支持的计算机和系统越来越复杂，它们很容易被看成是一些难懂的符咒，这些符咒只代表它们自己而看不出有什么含义，并且它们只是偶尔才能正常工作。

但是计算机编程不单是与程序相关，重要的是程序员要表达的思想。程序只是思想的静态表示，是曾经存在于程序员脑海中的某个结构的快照。程序是因为有了含义才值得写下来。那么是什么把代码和它的含义连接在一起呢？除了说“它做了该做的事”，怎样才能将一个程序的含义说得更具体一点呢？本章，你将会看到一些确定计算机程序含义的方法，了解如何给那些死板的“静态快照”注入生命气息。

2.1 “含义”的含义

在语言学中，*语义学*（semantics）研究的是单词和它们含义之间的关系：单词“dog”是纸上一些符号的组合，或是由某个人声带引起的一系列空气振动，这与真正的狗或者通常意义上狗的概念极为不同。语义不止关注抽象含义本身的基本性质，还关注具体的记号如何与它们的抽象含义关联起来。

计算机科学里，*形式语义学*注重找到确定程序难以捉摸的含义的方法，并利用这些方法发现或者证明编程语言中有趣的东西。形式语义学得到了广泛应用，从定义新的语言和进行编译优化这种具体的应用，到构造程序正确性的数学证明这样更抽象的领域不一而足。

为了完整地定义编程语言，我们需要：*语法*，描述程序看起来是什么样的；*语义*（semantics）[1]，描述程序的含义。

许多语言都没有官方的书面规范，而只有一个可用的解释器或者编译器。Ruby 本身算是“靠实现规范”这一类：尽管有很多关于 Ruby 应该如何工作的书和教程，但这些资料的最终源头都是松本行弘先生（Matz）的 Ruby 解释器（MRI，Matz’s Ruby Interpreter），这是 Ruby 的参考实现。如果任何一份 Ruby 文档与 MRI 的实际行为不一致，那必然是文档错了；JRuby、Rubinius 以及 MacRuby 这些第三方实现都只能努力地精准模拟 MRI 的行为，只有如此，它们才可以声称自己与 Ruby 语言有效地兼容。其他像 PHP 和 Perl 5 这样的语言，也使用了这种以实现为主导的语言定义方法。

另一种描述编程语言的方法，就是写一份平实的官方规范（一般是英语的）。C++、Java 以及 ECMAScript（JavaScript 的标准版本）都使用了这种方法：这些语言的标准化通过由专家委员会写成的、与实现无关的文档来完成，而且会存在很多与这些标准兼容的实现。比起只是依赖于一个参考实现，用官方文档规范定义一种语言更为严谨：这样所做的设计决策更有可能是经过深思熟虑、进行理性选择之后的，而不是某一个特定实现的意外结果。但是，规范通常非常难懂，而且很难讲规范中是不是含有矛盾、疏漏和有歧义的地方。特别是一份英语规范没有形式化的方法可以进行推导，我们只能完整彻底地阅读规范，大量地思考，然后寄希望于这样就可以掌握所有的前因后果。



Ruby 1.8.7 的规范确实存在，甚至已经被接受为 ISO 标准了（ISO/IEC 30170）[2]。尽管 mruby 工程（https://github.com/mruby/mruby）尝试构建一份轻量级、嵌入式的 Ruby 实现，并且明确声明将与 ISO 标准而不是 MRI 兼容，但 MRI 仍然被认为是 Ruby 语言由实现定义的权威规范。

注 1：在讨论编程语言理论的环境下，单词 semantics 通常被当作单数对待：我们通过为语言赋予语义来描述这种语言的含义。

注 2：尽管访问 ISO/IEC 30170 需要支付费用，但这一规范的一份早期草案可以免费下载：http://ipa.go.jp/osc/english/ruby/。

第三种方法是使用形式语义学中的数学方法准确描述编程语言的含义。它的目标是不仅能用适合系统分析甚至*自动化分析*的格式写出规范，还能保证其完全没有歧义，这样就可以对规范是否一致、是否含有冲突，以及是否有疏漏进行全面检查。在介绍如何处理语法之后，我们将会看到语义规范的这些形式化方法。

2.2 语法

传统的计算机程序是长长的字符串。每一种编程语言都有一系列规则，描述在那种语言中什么样的字符串被认为是有效程序。这些规则定义了这种语言的*语法*。

通过语言的语法规则，我们能把像 `y = x + 1` 这样可能有效的程序与像 `>/;x:1@4` 这样毫无意义的字符串区分开。语法规则还为如何阅读一些具有二义性的程序提供了有用信息，例如运算符优先级的规则能够自动判定 `1 + 2 * 3` 按其本意 `1 + (2 * 3)` 处理，而不是按 `(1 + 2) * 3` 处理。

当然，计算机程序的预期用途是被计算机读取，而要读程序就需要*语法解析器*：这个分析器程序能够读取代表程序的字符串，根据语法规则检查它是否有效，然后把它转换成一个适合被进一步处理的结构化表示。

有各种各样的工具能把一种语言的语法规则自动转换成一个语法解析器。具体如何对这些规则进行定义，以及把它们转成可用语法解析器的技术，并不是本章的讲解重点（2.6 节进行了简单介绍），但总体来讲一个语法解析器应该读入像 `y = x + 1` 这样的字符串，然后把它转换成*抽象语法树*（AST）。抽象语法树是源代码的一种表示，去掉了空格之类的无关细节，而只关注程序的分层结构。

语法关心的只是程序的表面是什么样的，而不是它的含义。程序有可能语法正确但没有任何实际意义。例如，程序 `y = x + 1` 本身可能没有任何意义，因为并没有事先说明 `x` 是什么，而程序 `z = true + 1` 可能会在运行时候报错，因为它试图在一个布尔型值上加数字。（当然，这依赖于具体编程语言的其他属性。）

正如我们所料，能说明如何把一种编程语言的语法与这个语法暗含的语义对应起来的“唯一正途”并不存在。实际上，关于程序的含义有几种不同的研究方法，它们都在形式化（formality）、抽象度（abstraction）、可表达性（expressiveness）和实际效率（efficiency）之间做了权衡。在接下来的几节里，我们将看到这些主要的形式化方法，并了解它们之间的联系。

2.3 操作语义

考虑程序含义的最实际方法是思考它*做了些什么*：在运行程序的时候，我们期望发生什么

呢？在运行时编程语言中不同的结构都是如何表现的？把它们放到一起组成更大的程序时会是什么效果？

这是*操作语义学*（operational semantic）的基础，这种方法为程序在某种机器上的执行定义一些规则，以此来捕捉编程语言的含义。这个机器常常是一种*抽象的机器*：为了解释这种语言所写的程序如何执行而设计出来的一个想象的、理想化的计算机。为了更好地捕获编程语言的运行时行为，通常需要针对不同种类的编程语言设计不同的抽象机器。

有了操作语义，我们可以朝着严谨而准确地研究语言中特定结构的目标前进了。用英语写成的语言规范可能暗藏着二义性，并且可能遗漏边缘情况，但一个形式化的操作性规范不会如此，为了令人信服地传达语言的行为，它必须明确而且无二义性。

2.3.1　小步语义

那么，我们如何设计一台抽象机器，并使用它定义一种编程语言的操作语义呢？一种方法就是假想一台机器，用这台机器直接按照这种语言的语法进行操作一小步一小步地对其进行反复*规约*，从而对一个程序求值。不管最后得到的结果含义是什么，我们每一步都能让程序更接近最终结果。

这种小步规约类似于对代数式求值的方式。例如，为了对 $(1 \times 2) + (3 \times 4)$ 求值，我们知道应该：

(1) 执行左侧的乘法（1×2 变成了 2），这样表达式就规约成了 $2 + (3 \times 4)$；
(2) 执行右侧的乘法（3×4 变成了 12），这样表达式规约成了 $2 + 12$；
(3) 执行加法（$2 + 12$ 变成了 14），最终得到 14。

我们可以认为 14 就是结果，因为通过上面步骤已经不能再进一步规约了；我们认为 14 是一个特殊代数表达式，它是一个*值*，有自己的含义，不需要进一步的努力了。

把如何进行每一小步的规约写成形式化规则，这个非形式化的过程就可以转换成一个操作语义。这些规则本身需要用某种语言（*元语言*）写下来，而这种语言通常是数学符号。

本章，我们将探索一个玩具级编程语言的语义，姑且将这种语言叫作 Simple[3]。

Simple 的小步语义（small-step semantic）的数学化描述如下所示：

注 3：你可以把它看成简单命令式语言（simple imperative language）的缩写。

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 \texttt{ + } e_2, \sigma\rangle \leadsto_e e_1' \texttt{ + } e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 \texttt{ + } e_2, \sigma\rangle \leadsto_e v_1 \texttt{ + } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ + } n_2, \sigma\rangle \leadsto_e n} \text{ if } n = n_1 + n_2$$

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 \texttt{ * } e_2, \sigma\rangle \leadsto_e e_1' \texttt{ * } e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 \texttt{ * } e_2, \sigma\rangle \leadsto_e v_1 \texttt{ * } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ * } n_2, \sigma\rangle \leadsto_e n} \text{ if } n = n_1 \times n_2$$

$$\frac{\langle e_1, \sigma\rangle \leadsto_e e_1'}{\langle e_1 \texttt{ < } e_2, \sigma\rangle \leadsto_e e_1' \texttt{ < } e_2} \qquad \frac{\langle e_2, \sigma\rangle \leadsto_e e_2'}{\langle v_1 \texttt{ < } e_2, \sigma\rangle \leadsto_e v_1 \texttt{ < } e_2'}$$

$$\frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \leadsto_e \texttt{true}} \text{ if } n_1 < n_2 \qquad \frac{}{\langle n_1 \texttt{ < } n_2, \sigma\rangle \leadsto_e \texttt{false}} \text{ if } n_1 \geq n_2$$

$$\frac{}{\langle x, \sigma\rangle \leadsto_e \sigma(x)} \text{ if } x \in \text{dom}(\sigma)$$

$$\frac{\langle e, \sigma\rangle \leadsto_e e'}{\langle x \texttt{ = } e, \sigma\rangle \leadsto_s \langle x \texttt{ = } e', \sigma\rangle} \qquad \frac{}{\langle x \texttt{ = } v, \sigma\rangle \leadsto_s \langle \texttt{do-nothing}, \sigma[x \mapsto v]\rangle}$$

$$\frac{\langle e, \sigma\rangle \leadsto_e e'}{\langle \texttt{if (}e\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \leadsto_s \langle \texttt{if (}e'\texttt{) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle}$$

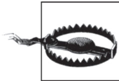
$$\frac{}{\langle \texttt{if (true) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \leadsto_s \langle s_1, \sigma\rangle} \qquad \frac{}{\langle \texttt{if (false) \{ }s_1\texttt{ \} else \{ }s_2\texttt{ \}}, \sigma\rangle \leadsto_s \langle s_2, \sigma\rangle}$$

$$\frac{\langle s_1, \sigma\rangle \leadsto_s \langle s_1', \sigma'\rangle}{\langle s_1\texttt{; }s_2, \sigma\rangle \leadsto_s \langle s_1'\texttt{; }s_2, \sigma'\rangle} \qquad \frac{}{\langle \texttt{do-nothing; }s_2, \sigma\rangle \leadsto_s \langle s_2, \sigma\rangle}$$

$$\frac{}{\langle \texttt{while (}e\texttt{) \{ }s\texttt{ \}}, \sigma\rangle \leadsto_s \langle \texttt{if (}e\texttt{) \{ }s\texttt{; while (}e\texttt{) \{ }s\texttt{ \} \} else \{ do-nothing \}}, \sigma\rangle}$$

从数学上讲，这是一个*推理规则*的集合，它定义了基于 Simple 抽象语法树的一个*规约关系*。实际点儿讲，这是一堆怪异的符号，关于计算机程序的含义它没有讲任何能让人理解的东西。

我们不会试图直接理解这种形式化的符号，而是研究如何用 Ruby 编写同样的推导规则。对程序员来说使用 Ruby 做元语言更容易理解，而且这样还有一个优点，就是这些规则可以执行，我们能看到它们是如何工作的。



我们并不打算尝试用“靠实现来规范”的方式描述 Simple 的语义。使用 Ruby 而不是用数学符号来描述小步语义，主要是为了使描述更容易被人们所理解。最终得到一个这种语言的可执行实现，只是这么做的额外好处。

使用 Ruby 有一大缺点：这是在使用一种更复杂的语言解释一种简单的语言，从哲学上来说这可能很失败。我们应该记住，数学化的规则是语义的权威描述，而使用 Ruby 只是为了更容易地理解这些规则的含义。

1. 表达式

首先来研究一下 Simple 语言中表达式的语义。规则将作用于这些表达式的抽象语法树，所以我们必须把 Simple 表达式表示成 Ruby 对象。要做到这一点，一种方式就是为 Simple 语法中每一种不同的元素都定义一个 Ruby 类，包括数字（number）、加法（add）、乘法（multiply）等，然后把每一个表达式表示成由这些类的实例构成的一棵树。

例如，下面是 Number、Add 和 Multiply 三个类的定义：

```
class Number < Struct.new(:value)
end

class Add < Struct.new(:left, :right)
end

class Multiply < Struct.new(:left, :right)
end
```

实例化这些类来手工构造抽象语法树：

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> #<struct Add
     left=#<struct Multiply
       left=#<struct Number value=1>,
       right=#<struct Number value=2>
     >,
     right=#<struct Multiply
       left=#<struct Number value=3>,
       right=#<struct Number value=4>
     >
   >
```



当然，最终我们想通过一个语法解析器自动构建这些树。2.6 节将介绍如何完成这件事情。

三个类（Number、Add 和 Multiply）都继承了 Struct 对 #inspect 的通用定义，所以在 IRB 中它们实例的字符串表示会含有大量不重要的细节。为了方便在 IRB 中查看抽象语法树的内容，我们将覆盖每个类的 #inspect 方法 [4]，让它返回自定义的字符串表示：

```
class Number
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
```

注 4：为了让代码保持简单，我们将抑制住把公共代码提取到超类或者模块中的欲望。

```
  end
end

class Add
  def to_s
    "#{left} + #{right}"
  end

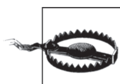
  def inspect
    "«#{self}»"
  end
end

class Multiply
  def to_s
    "#{left} * #{right}"
  end

  def inspect
    "«#{self}»"
  end
end
```

这样每个抽象语法树都将在 IRB 中以 Simple 源代码的形式呈现，外边会加上书名号（«»）以便与正常的 Ruby 值区分。

```
>> Add.new(
     Multiply.new(Number.new(1), Number.new(2)),
     Multiply.new(Number.new(3), Number.new(4))
   )
=> «1 * 2 + 3 * 4»
>> Number.new(5)
=> «5»
```



我们对 #to_s 的基本实现并没有把运算优先级考虑进来，所以有时候如果按照传统的优先级规则（例如 * 通常比 + 优先级更高）它们的输出是不正确的。以下面的抽象语法树为例：

```
>> Multiply.new(
     Number.new(1),
     Multiply.new(
       Add.new(Number.new(2), Number.new(3)),
       Number.new(4)
     )
   )
=> «1 * 2 + 3 * 4»
```

这棵树表示 «1 * (2 + 3) * 4» 与 «1 * 2 + 3 * 4» 不是一个表达式（具有不同的含义），但字符串表示并没有反映出这一点。

这个问题很严重，但与我们关于语义的讨论完全无关。为简单起见，暂时先忽略此事，避开可能拥有不正确字符串描述的表达式。我们将在 3.3.1 节为另一种语言给出更合适的实现。

现在为抽象语法树定义规约方法，这将是我们实现一个小步操作语义的起点。也就是说，代码可以以一个抽象语法树作为输入，然后生成一个规约树作为输出。

在实现规约本身之前，我们先要区分什么样的表达式能规约，什么样的表达式不能规约。Add 和 Multiply 表达式总是能规约的（它们的每一个表达式都表示一个操作，并能够通过那种操作对应的计算变成一个结果），但是 Number 表达式总是代表一个值，它就不能规约成任何其他东西了。

原则上，我们可以使用简单的 #reducible? 断言把这两种表达式区分开，它能判断参数是否可规约，并返回 true 或者 false：

```
def reducible?(expression)
  case expression
  when Number
    false
  when Add, Multiply
    true
  end
end
```



在 Ruby 的 case 语句里，控制表达式与 case 值是否匹配，是通过将控制表达式的值作为参数调用每个 case 值的 #=== 方法来判断的。方法 #=== 的实现会检查它的参数是否是那个类或者那个子类的实例，这样我们可以使用“case 对象 when 类名”这样的语法为一个类匹配一个对象。

但是，在一种面向对象语言里这么写代码通常被认为是不好的做法[5]；如果一些运算的行为依赖于它参数的类型，典型的做法是将这种每个类都有的行为实现为它们的实例方法，从而让语言隐式地决定调用哪个方法，而不是使用显式的 case 语句。

因此，我们将分别为 Number、Add 和 Multiply 实现 #reducible? 方法：

```
class Number
  def reducible?
    false
  end
end

class Add
  def reducible?
    true
  end
end

class Multiply
  def reducible?
```

注 5：尽管我们用 Haskell 或者 ML 这样的函数式语言写 #reducible? 时就是这么写的。

```
    true
  end
end
```

这回的表现正是我们想要的：

```
>> Number.new(1).reducible?
=> false
>> Add.new(Number.new(1), Number.new(2)).reducible?
=> true
```

现在可以为这些表达式实现规约了：像上面一样，我们为 Add 和 Multiply 定义一个 #reduce 方法。既然数字不能再规约，那就没有必要定义 Number#reduce 了，因此除非确切知道一个表达式能够规约，否则不要对其调用 #reduce 方法。

那么规约加法表达式的规则是什么呢？如果左右参数都是数字，那我们就能把它们加到一起，但如果其中一个或者所有参数需要规约怎么办？既然我们在考虑一小步一小步地进行规约，那就有必要在它们都符合规约条件的时候决定哪个参数先进行规约[6]。一个常用的策略是按照从左到右的顺序对参数进行规约，规则是这样的：

- 如果加法左边的参数能够规约，就规约左边的参数；
- 如果加法左边的参数不能规约，但是右边的参数可以规约，就规约右边的参数；
- 如果两边都不能规约，它们应该都是数字了，就把它们加到一起。

上面这些规则的结构是小步规约操作语义的特征。每一个规则都提供了它能得以应用的表达式模式（左边参数可规约的加法，右边参数可规约的加法，两边参数分别都不能规约的加法），还有对当模式匹配上之后如何构建一个规约后的新表达式的描述。选择了这些特定的规则之后，我们不仅确定了那些参数分别规约好之后应该如何合并到一起，还特别指出了一个 Simple 表达式要使用从左到右求值的方法对参数进行规约。

我们可以把这些规则直接翻译成一个 Add#reduce 的实现，同样的代码对 Multiply#reduce 也适用（别忘了要把参数乘起来而不是加起来）：

```
class Add
  def reduce
    if left.reducible?
      Add.new(left.reduce, right)
    elsif right.reducible?
      Add.new(left, right.reduce)
    else
      Number.new(left.value + right.value)
    end
  end
end
```

注 6：选择什么顺序并没有区别，但是在这个时候我们必须做出决策。

```
class Multiply
  def reduce
    if left.reducible?
      Multiply.new(left.reduce, right)
    elsif right.reducible?
      Multiply.new(left, right.reduce)
    else
      Number.new(left.value * right.value)
    end
  end
end
```



方法 #reduce 总是构建出新的表达式，而不是对已有的表达式进行修改。

为这几种表达式实现了 #reduce 方法之后，我们可以反复对其进行调用，从而通过很多的一小步来完整地求出表达式的值：

```
>> expression =
     Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
=> «1 * 2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 3 * 4»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «2 + 12»
>> expression.reducible?
=> true
>> expression = expression.reduce
=> «14»
>> expression.reducible?
=> false
```



注意，#reduce 总是把一个表达式转换成另一个表达式，这正是小步规约操作语义应该遵守的规则。特别要注意的是，Add.new(Number.new(2),Number.new(12)).reduce 返回的 Number.new(14) 表示 Simple 表达式，而不仅仅是 14 这个 Ruby 中的数字。

Simple 语言（我们正在为其定义语义）和 Ruby 元语言（我们正在使用它定义语义）在明显不同的时候区分起来很容易——就像元语言是数学符号而不是一种程序设计语言时一样容易区分——但是这里因为两种语言看起来很像，所以需要更加小心。

我们在维护着一个状态——也就是当前表达式——并且对其反复调用 #reducible? 和 #reduce，直到得到了一个值为止，通过这种方式，可以手工模拟一个抽象机器对表达式求值的操作。为了节省点力气，也为了让这个抽象机器的思想更为具体，我们可以轻松地写些 Ruby 代码。把这些代码和状态封装到一个类里，并称为虚拟机：

```
class Machine < Struct.new(:expression)
  def step
    self.expression = expression.reduce
  end

  def run
    while expression.reducible?
      puts expression
      step
    end
    puts expression
  end
end
```

这允许我们用一个表达式实例化一个虚拟机，让它运行（#run），并观察逐渐规约的各个步骤：

```
>> Machine.new(
     Add.new(
       Multiply.new(Number.new(1), Number.new(2)),
       Multiply.new(Number.new(3), Number.new(4))
     )
   ).run
1 * 2 + 3 * 4
2 + 3 * 4
2 + 12
14
=> nil
```

要扩展这个实现以支持其他简单的值和运算并不难：减法和除法，布尔值 true 和 false，布尔运算 and、or 和 not，对数字进行比较并返回布尔值的运算，等等。例如，下面是一个布尔值以及小于运算的实现：

```
class Boolean < Struct.new(:value)
  def to_s
    value.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    false
  end
end
```

```
class LessThan < Struct.new(:left, :right)
  def to_s
    "#{left} < #{right}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce
    if left.reducible?
      LessThan.new(left.reduce, right)
    elsif right.reducible?
      LessThan.new(left, right.reduce)
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

这仍然允许我们一小步一小步地规约布尔表达式：

```
>> Machine.new(
     LessThan.new(Number.new(5), Add.new(Number.new(2), Number.new(2)))
   ).run
5 < 2 + 2
5 < 4
false
=> nil
```

目前为止都是直截了当的东西：我们通过实现能对一种语言求值的虚拟机来定义它的操作语义。虚拟机当前的状态就是当前的表达式，而机器的行为是由一个规则集合来描述的，这个规则集合负责管理机器运行时的状态切换。我们已经把机器实现成了程序，这个程序跟踪当前表达式，持续对其进行规约，并随之更新表达式，直到没有更进一步的规约可以继续执行为止。

但是这种由简单代数表达式组成的语言不是十分有趣，这种语言没有几个我们期望拥有的哪怕是最简单编程语言中的特性。接下来我们把它构建得更复杂一些，让它看起来更像是一种能写出有用程序的语言。

首先，Simple 有一个明显缺失的东西：变量。在任何有用的语言中，我们都期望在讨论值时能够使用有意义的名字而不是它们本身的字面值。这些名字提供了一个间接层，这样同一个代码可以用来处理很多不同的值——包括来自于程序外部因而在写代码时甚至都不知道的值。

我们可以引入一个新的表达式类 Variable 来表示 Simple 中的变量：

```
class Variable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end
end
```

为了能规约一个变量，抽象机器不仅仅需要存储当前表达式，还要存储从变量名称到它们值的映射——环境（environment）。在 Ruby 中，我们可以把这个映射实现成一个散列表（hash），其中用符号作为键，用表达式对象作为值；例如，散列表 {x:Number.new(2), y:Boolean.new(false) } 是一个环境，它分别把变量 x 和 y 与 Simple 的数字和布尔值进行了关联。



对这种语言来说，环境的目的只是把变量名映射到 Number.new(2) 这样不可规约的值上，而不是映射到 Add.new(Number.new(1), Number.new(2)) 这样可以规约的表达式。稍后我们编写能改变环境的规则时要注意这个约束。

有了环境，我们很容易实现 Variable#reduce：它只是在环境里查找变量的名字并返回其值。

```
class Variable
  def reduce(environment)
    environment[name]
  end
end
```

注意，我们正在把一个环境作为参数传进 #reduce，所以需要修改其他类的 #reduce 的实现，以便能接受和提供这个参数：

```
class Add
  def reduce(environment)
    if left.reducible?
      Add.new(left.reduce(environment), right)
    elsif right.reducible?
      Add.new(left, right.reduce(environment))
    else
      Number.new(left.value + right.value)
    end
  end
end
```

```
class Multiply
  def reduce(environment)
    if left.reducible?
      Multiply.new(left.reduce(environment), right)
    elsif right.reducible?
      Multiply.new(left, right.reduce(environment))
    else
      Number.new(left.value * right.value)
    end
  end
end

class LessThan
  def reduce(environment)
    if left.reducible?
      LessThan.new(left.reduce(environment), right)
    elsif right.reducible?
      LessThan.new(left, right.reduce(environment))
    else
      Boolean.new(left.value < right.value)
    end
  end
end
```

现在 #reduce 的所有实现在更新之后都已经能支持环境了，因此还需要重新定义虚拟机，以便维持一个环境并把它提供给 #reduce：

```
Object.send(:remove_const, :Machine) # 忘记原来的 Machine 类

class Machine < Struct.new(:expression, :environment)
  def step
    self.expression = expression.reduce(environment)
  end

  def run
    while expression.reducible?
      puts expression
      step
    end

    puts expression
  end
end
```

机器对 #run 的定义仍然没变，但它有了一个新的环境属性，这个属性提供给 #step 方法新的实现使用。

现在只要我们也提供一个包含变量值的环境，就可以对包含变量的表达式进行规约了：

```
>> Machine.new(
     Add.new(Variable.new(:x), Variable.new(:y)),
     { x: Number.new(3), y: Number.new(4) }
   ).run
```

```
x + y
3 + y
3 + 4
7
=> nil
```

环境的引入完成了表达式的操作语义。我们已经设计了抽象机器，它由一个初始表达式和环境开始，然后在每次规约的一小步中使用当前的表达式和环境生成一个新的表达式，这个过程中环境始终没有改变。

2. 语句

现在我们可以看一下另一种程序结构的实现：语句。它是一个表达式，用来求值生成另一个表达式；换句话说，一个语句能够通过求值改变抽象机器的状态。机器唯一的状态（除了当前程序）就是环境，因此我们将允许 Simple 的语句生成一个新的环境以替换当前环境。

最简单的语句就是什么都不做的语句：它不能规约，因为对环境没有任何影响。这实现起来很简单：

```
class DoNothing ❶
  def to_s
    'do-nothing'
  end

  def inspect
    "«#{self}»"
  end

  def ==(other_statement) ❷
    other_statement.instance_of?(DoNothing)
  end

  def reducible?
    false
  end
end
```

❶ 其他所有语法类都从 Struct 类继承，但是 DoNothing 没有继承任何类。这是因为 DoNothing 什么属性都没有，而且遗憾的是，Struct.new 还不让我们传一个空的属性名称列表。

❷ 想要比较任意两个语句是否相等。其他类都从 Struct 继承了 #== 的实现，但 DoNothing 只能定义它自己的了。

一个什么都不做的语句可能看起来没什么意义，但是能有一个特殊的语句表示程序已经执行成功会非常方便。其他语句完成了它们的工作之后，我们会将它们最终规约成 «do-nothing»。

要看个实用语句的例子，最简单的就是像 «x = x + 1» 这样的赋值语句，但在实现赋值语句之前，我们还需要决定它的规约规则。

一个赋值语句由一个变量名（x）、一个等号和一个表达式（«x + 1»）组成。如果赋值语句中的表达式是可规约的，我们就可以按照表达式规约规则对其进行规约并最终得到一个包含规约后表达式的新的赋值语句。例如，在一个变量 x 值为 «2» 的环境里对 «x = x + 1» 进行规约，我们会得到语句 «x = 2 + 1»，然后再把它规约就得到 «x = 3»。

可是然后呢？如果表达式已经是 «3» 这样的值了，那么我们就应该执行赋值，也就意味着对环境进行更新，即把这个值与适当的变量名关联起来。因此规约一个语句不单需要生成一个规约了的新语句，还要产生一个新的环境，这个环境有时候会与执行规约时的环境不同。



我们的实现将使用 Hash#merge 创建一个新的散列来更新环境，不会改变旧值：

```
>> old_environment = { y: Number.new(5) }
=> {:y=>«5»}
>> new_environment = old_environment.merge({ x: Number.new(3) })
=> {:y=>«5», :x=>«3»}
>> old_environment
=> {:y=>«5»}
```

可以选择破坏性地改变当前环境，而不是创建一个新的，但是避免破坏性的修改可以促使我们把 #reduce 的结果完全明确出来。如果 #reduce 想要改变当前的环境，它就得给调用者返回一个改变后的环境进行通知；反之，如果它不返回一个环境，那么就可以肯定没有造成任何变化。

这个约束帮助我们强化了表达式和语句的区别。对于表达式，把一个环境传递给 #reduce，然后得到一个规约了的表达式；因为没有返回一个新的环境，所以很明显规约一个表达式不会改变环境。对于语句，我们将用当前的环境调用 #reduce，然后得到一个新的环境，这表明规约一个语句会对环境有影响。（换句话说，Simple 小步语义的结构告诉我们：Simple 的表达式是纯净无害的，而它的语句不是这样。）

因此从一个空的环境规约 «x = 3» 应该会产生一个新的环境 { x: Number.new(3) }，但是我们还期望这个语句以某种方式得到规约；不然的话，抽象机器将会不断地把 «3» 赋值给 x。这时候 «do-nothing» 就派上用场了：一个完整的赋值语句规约成 «do-nothing»，就表明语句的规约已经结束，并且可以认为新环境中的东西就是执行结果。

总结起来，赋值的规约规则是：

- 如果赋值表达式能规约，那么就对其规约，得到的结果就是一个规约了的赋值语句和一个没有改变的环境；
- 如果赋值表达式不能规约，那么就更新环境把这个表达式与赋值的变量关联起来，得到的结果是一个 «do-nothing» 语句和一个新的环境。

这样，我们就有了实现一个赋值类 Assign 的足够信息。唯一的困难就是 Assign#reduce 需要既返回一个语句又返回一个环境——而 Ruby 的方法只能返回一个对象——但我们可以把它们放到由两个元素组成的数组中返回，这就模拟了这种情况。

```
class Assign < Struct.new(:name, :expression)
  def to_s
    "#{name} = #{expression}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if expression.reducible?
      [Assign.new(name, expression.reduce(environment)), environment]
    else
      [DoNothing.new, environment.merge({ name => expression })]
    end
  end
end
```



正如我们承诺的那样，Assign 的规约规则保证了如果一个表达式不可规约（如一个值），它就只会增加到环境上。

可以像表达式一样对一个赋值语句反复规约，直到其不能再规约为止。通过这个方法就可以对一个赋值表达式求值。

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> environment = { x: Number.new(2) }
=> {:x=>«2»}
>> statement.reducible?
=> true
>> statement, environment = statement.reduce(environment)
=> [«x = 2 + 1», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«x = 3», {:x=>«2»}]
>> statement, environment = statement.reduce(environment)
=> [«do-nothing», {:x=>«3»}]
```

```
>> statement.reducible?
=> false
```

这个过程甚至比手工规约表达式更难，因此为了处理语句，需要重新实现虚拟机，让它能在每一步规约时显示当前的语句和环境：

```
Object.send(:remove_const, :Machine)

class Machine < Struct.new(:statement, :environment)
  def step
    self.statement, self.environment = statement.reduce(environment)
  end

  def run
    while statement.reducible?
      puts "#{statement}, #{environment}"
      step
    end

    puts "#{statement}, #{environment}"
  end
end
```

现在这台机器又可以为我们工作啦：

```
>> Machine.new(
     Assign.new(:x, Add.new(Variable.new(:x), Number.new(1))),
     { x: Number.new(2) }
   ).run
x = x + 1, {:x=>«2»}
x = 2 + 1, {:x=>«2»}
x = 3, {:x=>«2»}
do-nothing, {:x=>«3»}
=> nil
```

可以看到，这台机器仍然在执行表达式的规约步骤（«x + 1» 规约成 «2 + 1»，再规约成 «3»），但是这个规约过程现在不是发生在语法树的顶层，而是在一个语句里。

既然知道语句规约是如何工作的了，那么我们就可以对其进行扩展，以支持其他类型的语句。让我们从 «if (x) { y = 1 } else { y = 2 }» 这样的语句开始，这个语句包含了一个叫作条件（«x»）的表达式，还有两个语句，一个称为结果（«y = 1»），另一个是替代语句（«y = 2»）[7]。对条件进行规约的规则很简单：

- 如果条件能规约，那就对其进行规约，得到的结果是一个规约了的条件语句和一个没有改变的环境；
- 如果条件是表达式 «true» 了，就规约成结果语句和一个没有变化的环境；

注 7：此条件语句与 Ruby 的 if 不同，Ruby 中的 if 是返回一个值的表达式，但是在 Simple 中，这是一个语句，它从其他两个语句中选择一个求值，并且它唯一的结果就是对当前环境的影响。

• 如果条件是表达式 «false»，就规约成替代语句和一个没有变化的环境。

在这种情况下，所有规则都不会改变环境——第一条规则中对条件表达式的规约只会生成一个新的表达式，而不会产生新的环境。

下面是翻译成 If 类的规则：

```
class If < Struct.new(:condition, :consequence, :alternative)
  def to_s
    "if (#{condition}) { #{consequence} } else { #{alternative} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    if condition.reducible?
      [If.new(condition.reduce(environment), consequence, alternative), environment]
    else
      case condition
      when Boolean.new(true)
        [consequence, environment]
      when Boolean.new(false)
        [alternative, environment]
      end
    end
  end
end
```

下面是规约操作：

```
>> Machine.new(
     If.new(
       Variable.new(:x),
       Assign.new(:y, Number.new(1)),
       Assign.new(:y, Number.new(2))
     ),
     { x: Boolean.new(true) }
   ).run
if (x) { y = 1 } else { y = 2 }, {:x=>«true»}
if (true) { y = 1 } else { y = 2 }, {:x=>«true»}
y = 1, {:x=>«true»}
do-nothing, {:x=>«true», :y=>«1»}
=> nil
```

这些都与预期一致，但如果能支持不带 «else» 从句的条件语句就好了，比如 «if (x) {y = 1}»。幸运的是，把语句写成 «if (x) { y = 1 } else { do-nothing }» 就可以做到，这和

没有 «else» 从句的效果是一样的：

```
>> Machine.new(
     If.new(Variable.new(:x), Assign.new(:y, Number.new(1)), DoNothing.new),
     { x: Boolean.new(false) }
   ).run
if (x) { y = 1 } else { do-nothing }, {:x=>«false»}
if (false) { y = 1 } else { do-nothing }, {:x=>«false»}
do-nothing, {:x=>«false»}
=> nil
```

既然不仅实现了表达式，还实现了赋值语句和条件语句，我们就有了组成程序所需要的基础材料，这样的程序可以执行计算和进行决策，做实际的工作。主要的限制是我们还不能把这些基础材料“连接”到一起：没有办法给多个变量赋值或者执行多个条件运算，这大幅度地限制了语言的可用性。

为摆脱这个限制我们可以再定义一种语句——*序列*（sequence），它把两个语句（如 «x = 1 + 1» 和 «y = x + 3»）连接到一起，组成一个更大的语句（如 «x = 1 + 1; y = x + 3»）。一旦有了序列语句，我们就可以反复使用它们构建更大的语句；例如，序列 «x = 1 + 1; y = x + 3» 和赋值语句 «z = y + 5» 能连到一起组成序列 «x = 1 + 1; y = x + 3; z = y + 5»[8]。

对序列进行规约的规则有点微妙：

- 如果第一条语句是 «do-nothing»，就规约成第二条语句和原始的环境；
- 如果第一条语句不是 «do-nothing»，就对其进行规约，得到的结果是一个新的序列（规约之后的第一条语句，后边跟着第二条语句）和一个规约了的环境。

看了代码你会更清楚这些规则：

```
class Sequence < Struct.new(:first, :second)
  def to_s
    "#{first}; #{second}"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    case first
    when DoNothing.new
```

注 8：为了达到我们的目的，这个语句构造成 «(x = 1 + 1; y = x + 3); z = y + 5» 还是 «x = 1 + 1; (y = x + 3; z = y + 5)» 都没有关系。在执行规约时，这个选择会影响规约的顺序，但是两种方式最终的结果是一样的。

```
      [second, environment]
    else
      reduced_first, reduced_environment = first.reduce(environment)
      [Sequence.new(reduced_first, second), reduced_environment]
    end
  end
end
```

这些规则的总体效果就是：不断规约一个序列时，一直都在规约它的第一个语句，直到成为 «do-nothing»，然后再去规约第二个语句。在虚拟机里运行一个序列，我们可以看到这种效果：

```
>> Machine.new(
     Sequence.new(
       Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
       Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
     ),
     {}
   ).run
x = 1 + 1; y = x + 3, {}
x = 2; y = x + 3, {}
do-nothing; y = x + 3, {:x=>«2»}
y = x + 3, {:x=>«2»}
y = 2 + 3, {:x=>«2»}
y = 5, {:x=>?2?}
do-nothing, {:x=>«2», :y=>«5»}
=> nil
```

Simple 里重要但仍缺失的只有某种无限制的循环结构了，所以为了完成任务，我们引入一个 «while» 语句，以便程序可以执行任意次数的重复计算[9]。像 «while(x < 5) { x = x * 3» 这样的语句，包含了一个叫作*条件*（«x < 5»）的表达式和一个叫作*语句主体*（body）的语句（«x = x * 3»）。

为一个 «while» 语句写出正确的规约规则需要一点技巧。我们尝试着像 «if» 语句那样对其处理：如果能规约就对条件进行规约；不能的话，就根据条件是 «true» 还是 «false» 相应地规约语句主体或者执行 «do-nothing»，那下一步会怎么样呢？条件已经被规约成一个值或者丢弃了，并且语句主体已经被规约成 «do-nothing»，那么我们如何执行下一周期的循环呢？每一步规约要想与将来的规约步骤交流，只能通过产生一个新的语句和环境来实现，而使用这种方法，我们就没有地方记录最初的条件和语句主体供下一个循环使用。

小步的解决方式[10]是使用序列语句把 «while» 的一个级别*展开*，把它规约成一个只执行一次循环的 «if» 语句，然后再重复原始的 «while»。这意味着我们只需要一个规约规则：

注 9：使用序列语句，我们已经能够硬编码固定数量的重复操作了，但还是无法控制运行时的重复行为。

注 10：我们总试图把 «while» 的迭代行为直接构建成规约规则，而不是找到一种途径让抽象机器去处理它，但这不是小步语义的工作方式。参考 2.3.2 节，其中介绍的大步语义是一种让规则完成工作的语义。

• 把 «while （条件） { 语句主体 }» 规约成 «if （条件） { 语句主体； while （条件） { 语句主体 } } else { do-nothing }» 和一个没有改变的环境。

在 Ruby 中实现这个规则很容易：

```
class While < Struct.new(:condition, :body)
  def to_s
    "while (#{condition}) { #{body} }"
  end

  def inspect
    "«#{self}»"
  end

  def reducible?
    true
  end

  def reduce(environment)
    [If.new(condition, Sequence.new(body, self), DoNothing.new), environment]
  end
end
```

这给了虚拟机根据需要对条件和语句主体进行求值的机会：

```
>> Machine.new(
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     ),
     { x: Number.new(1) }
   ).run
while (x < 5) { x = x * 3 }, {:x=>«1»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (1 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«1»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 1 * 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
x = 3; while (x < 5) { x = x * 3 }, {:x=>«1»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«3»}
while (x < 5) { x = x * 3 }, {:x=>«3»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (3 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
if (true) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«3»}
x = x * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 3 * 3; while (x < 5) { x = x * 3 }, {:x=>«3»}
x = 9; while (x < 5) { x = x * 3 }, {:x=>«3»}
do-nothing; while (x < 5) { x = x * 3 }, {:x=>«9»}
while (x < 5) { x = x * 3 }, {:x=>«9»}
if (x < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
if (9 < 5) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
if (false) { x = x * 3; while (x < 5) { x = x * 3 } } else { do-nothing }, {:x=>«9»}
do-nothing, {:x=>«9»}
=> nil
```

或许这个规约规则看起来有点像是在逃避——好像我们总是在往后推迟对 «while» 的规约，一直没有实际进展——但它确实很好地解释了一个 «while» 语句真正的意思：检查条件，对语句主体求值，然后重新开始。奇怪的是，对 «while» 进行规约，会把它转换成一个语法上更庞大的程序，其中包括条件语句和序列语句，而不是直接对它的条件和语句主体进行规约，但有一个能定义一种语言形式语义的技术方案是非常好的，因为我们会更易理解这种语言中的不同部分彼此之间是如何关联的。

3. 正确性

如果程序只是语法有效但实际上是错误的，这时按照我们给出的语义执行会发生什么呢？我们之前完全忽视了这一点。语句 «x = true; x = x + 1» 是一段语法有效的 Simple 代码，我们确实可以构建一个抽象语法树来表示它，但试图反复对其规约的时候，它将会崩溃，因为在尝试往 «true» 上加 «1» 的时候抽象机器会终止。

```
>> Machine.new(
     Sequence.new(
       Assign.new(:x, Boolean.new(true)),
       Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
     ),
     {}
   ).run
x = true; x = x + 1, {}
do-nothing; x = x + 1, {:x=>«true»}
x = x + 1, {:x=>«true»}
x = true + 1, {:x=>«true»}
NoMethodError: undefined method `+' for true:TrueClass
```

处理这个问题的一个方法就是在表达式能被规约的时候增加更多的约束，加入对求值失败可能性的考虑，这时求值过程有可能会中止，而不是总要试图规约成一个值（然后就可能在处理过程中崩溃）。我们本来可以把 Add#reducible? 实现成这样：«+» 的两个参数要么都是可规约的，要么都是数字类型（Number）实例，这时它才返回 true，这种情况下，表达式 «true + 1» 将会中止处理而永远不会变成一个值。

最终，我们需要一个比语法更强大的工具，它要能“看到未来”并让我们避免执行任何可能崩溃或者中止处理的程序。这一章是关于*动态语义*（dynamic semantic）的——程序执行时具体在做什么——但那并不是一个程序所拥有的唯一一种含义；在第 9 章，我们将研究*静态语义*（static semantic），看看如何根据语言的动态语义来判断一个语法上有效的程序是否具有有用的含义。

4. 应用

我们定义的程序设计语言非常基本，但在写下所有规约规则的时候，仍然不得不做了一些设计上的决策并明确地表述它们。例如，与 Ruby 不同的是，Simple 这种语言会区分表达式和语句，前者返回一个值，后者不会返回值；与 Ruby 相同的是，Simple 的环境只与已

经完全规约成值的变量关联，而不与仍然有待执行的更大表达式关联[11]。我们可以通过给出不同的小步语义来改变上面任何的策略，这将描述一种新的语言，这种语言拥有同样的语法，但有着不同的运行时行为。如果向语言中增加更多精心设置的特性——数据结构、过程调用、异常和一个对象系统——我们需要做出更多的设计决策并在定义语义时无歧义地表达它们。

小步语义的细节化、面向执行的风格能让它无歧义地定义真实世界的编程语言。例如，Scheme 编程语言最新的 R6RS 标准使用了小步语义（http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-15.html）描述其执行，并提供了 PLT Redex 语言（http://redex.racket-lang.org/）（设计用来定义和调试操作语义的一门特定领域的语言）对那些语义的参考实现（http://www.r6rs.org/refimpl）。OCaml 编程语言，在一个更简单的 Core ML 语言基础之上构建了一系列的分层，也有对于基础语言运行时行为的小步语义定义（http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc5）。

参考 6.2.2 节，那里还有一个小步操作语义的例子，它用了一个甚至更简单的叫作 lambda 演算的编程语言定义了表达式的含义。

2.3.2　大步语义

我们已经看到了小步操作语义是什么样子的：设计一台抽象机器维护一些执行状态，然后定义一些规约规则，这些规则详细说明了如何才能对每种程序结构循序渐进地求值。特别地，小步语义大部分都带有迭代的味道，它要求抽象机器反复执行规约步骤（`Machine#run` 中的 `while` 循环），这些步骤以及与它们同样类型的信息可以作为自身的输入和输出，这让它们适合这种反复进行的应用程序。[12]

这种小步的方法有一个优势，就是能把执行程序的复杂过程分成更小的片段解释和分析，但它确实有点不够直接：我们没有解释整个程序结构是如何工作的，而只是展示了它是如何慢慢规约的。为什么不能更直接地解释一个语句，完整地说明它的执行过程呢？好吧，我们可以，而这正是*大步语义*（big-step semantic）的依据。

大步语义的思想是，定义如何从一个表达式或者语句直接得到它的结果。这必然需要把程序的执行当成一个*递归*的而不是迭代的过程：大步语义说的是，为了对一个更大的表达式求值，我们要对所有比它小的子表达式求值，然后把结果结合起来得到最终答案。

在很多方面，这都比小步的方法更自然，但确实失去了一些对细节的关注。例如，小步语义明确定义了操作应该发生的顺序，因为在每一步都明确了下一步规约应该是什么。但是

注 11：Ruby 的 proc 在某种意义上允许把复合表达式复制给变量，但是一个 proc 仍然是一个值：它本身不能再执行任何求值操作了，但是能和其他值一起作为一个更大表达式的一部分进行规约。

注 12：对一个表达式和一个环境进行规约将得到一个新的表达式，而且下一次还可以重用旧的环境；对一个语句和一个环境进行规约将得到一个新的语句和一个新的环境。

大步语义经常会写成更为松散的形式，只会说哪些子计算会执行，而不会指明它们按什么顺序执行。[13] 小步语义还提供一种轻松的方式用以监视计算的中间阶段，而大步语义只是返回一个结果，不会产生任何关于如何计算的证据。

为了理解做出的这种权衡，让我们回顾一些常见的语言结构，并看如何在 Ruby 中实现它们的大步语义。我们的小步语义要求有一个 `Machine` 类跟踪状态并反复执行规约，但是这里不需要这个类了；大步规约的规则描述了如何只对程序的抽象语法树访问一次就计算出整个程序的结果，因此不需要处理状态和重复。我们将只对表达式和语句类定义一个 `#evaluate` 方法，然后直接调用它。

1. 表达式

处理小步语义时，我们不得不区分像 «1 + 2» 这样可规约的表达式和像 «3» 这样不可规约的表达式，这样规约规则才能识别一个子表达式什么时候可以用来组成更大的程序。但是在大步语义中，每个表达式都能求值。唯一的区别，如果我们想要有个区别的话，就是对一些表达式求值会直接得到它们自身，而对另一些表达式求值会执行一些计算并得到一个不同的表达式。

大步语义的目标是像小步语义那样对一些运行时行为进行建模，这意味着我们期望对于每一种程序结构，大步语义规则都要与小步语义规则程序最终生成的东西保持一致。（把操作语义写成数学形式之后，这是能被准确证明的。）小步语义规则规定，像数值（`Number`）和布尔值（`Boolean`）这样的值不能再规约了，因此它们的大步规约非常简单：求值的结果直接就是它们本身。

```
class Number
  def evaluate(environment)
    self
  end
end

class Boolean
  def evaluate(environment)
    self
  end
end
```

变量（`Variable`）表达式是唯一的，这样它们的小步语义允许它们在成为一个值之前只规约一次，所以它们的大步语义规则与小步规则一样：在环境中查找变量名然后返回它的值。

```
class Variable
  def evaluate(environment)
    environment[name]
```

注 13：我们用这种方法实现的大步语义不会有二义性，因为 Ruby 本身已经进行了排序决策，但是在数学化地定义大步语义时，就不可避免地要讲清楚准确的求值策略了。

```
  end
end
```

二元表达式 Add、Multiply 和 LessThan 更有意思，它们要求先对左右子表达式递归求值，然后再用恰当的 Ruby 运算合并两边的结果值：

```
class Add
  def evaluate(environment)
    Number.new(left.evaluate(environment).value + right.evaluate(environment).value)
  end
end

class Multiply
  def evaluate(environment)
    Number.new(left.evaluate(environment).value * right.evaluate(environment).value)
  end
end

class LessThan
  def evaluate(environment)
    Boolean.new(left.evaluate(environment).value < right.evaluate(environment).value)
  end
end
```

为了检查这些大步的表达式语义是否正确，下面将在 Ruby 的控制台验证一下：

```
>> Number.new(23).evaluate({})
=> «23»
>> Variable.new(:x).evaluate({ x: Number.new(23) })
=> «23»
>> LessThan.new(
     Add.new(Variable.new(:x), Number.new(2)),
     Variable.new(:y)
   ).evaluate({ x: Number.new(2), y: Number.new(5) })
=> «true»
```

2. 语句

在我们要定义语句的行为时，这种类型的语义就能发挥作用了。在小步语义下表达式会规约成其他表达式，但语句会规约成 «do-nothing» 并且得到一个经过修改的环境。我们可以把大步语义的语句求值看成一个过程，这个过程总是把一个语句和一个初始环境转成一个最终的环境，这避免了小步语义不得不对 #reduce 产生的中间语句进行处理的复杂性。例如，对一个赋值语句按照大步的方法求值应该完整地对其表达式求值，并返回一个包含结果值的更新了的环境：

```
class Assign
  def evaluate(environment)
    environment.merge({ name => expression.evaluate(environment) })
  end
end
```

类似地，DoNothing#evaluate 无疑将把未更改的环境返回，而 If#evaluate 的工作相当地直

接：对条件求值，然后把环境返回，这个环境来自于对序列或者替代语句求值得到的结果。

```
class DoNothing
  def evaluate(environment)
    environment
  end
end

class If
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      consequence.evaluate(environment)
    when Boolean.new(false)
      alternative.evaluate(environment)
    end
  end
end
```

有两种有趣的情况就是序列语句和 «while» 循环表达式。对于序列，我们只需要对两个语句求值，但是初始环境需要“穿过”这两个求值过程，这样第一个语句求值的结果就能成为第二个语句求值的环境。这可以写成 Ruby 代码：用第一次求值的结果作为第二次求值的参数：

```
class Sequence
  def evaluate(environment)
    second.evaluate(first.evaluate(environment))
  end
end
```

为了让先前的语句为后边的做准备，“穿过”环境是至关重要的：

```
>> statement =
     Sequence.new(
       Assign.new(:x, Add.new(Number.new(1), Number.new(1))),
       Assign.new(:y, Add.new(Variable.new(:x), Number.new(3)))
     )
=> «x = 1 + 1; y = x + 3»
>> statement.evaluate({})
=> {:x=>«2», :y=>«5»}
```

对于 «while» 语句，我们需要彻底想清楚对一个循环完整求值的各个阶段：

- 对条件求值，得到 «true» 或者 «false»；
- 如果条件求值结果是 «true»，就对语句主体求值得到一个新的环境，然后在那个新的环境下重复循环（也就是说对整个 «while» 语句再次求值），最后返回作为结果的环境；
- 如果条件求值结果是 «false»，就返回未修改的环境。

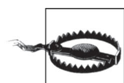
这是对一个 «while» 语句行为的递归解释。就像序列语句，循环体生成的更新了的环境被

下一个迭代使用这一点非常重要；不然的话，条件一直都是 «true»，那么循环就永远也没有机会停下来了。[14]

知道了大步 «while» 语义的行为表现之后，就可以实现 While#evaluate 了：

```
class While
  def evaluate(environment)
    case condition.evaluate(environment)
    when Boolean.new(true)
      evaluate(body.evaluate(environment)) ❶
    when Boolean.new(false)
      environment
    end
  end
end
```

❶ 循环在这里发生：body.evaluate(environment) 对循环求值得到一个新的环境，然后我们把那个环境传回当前方法中开始下一次迭代。这意味着可能会堆积很多对 While#evaluate 的嵌套调用，直到条件最后成为 «false» 然后返回最后的环境。



就像任何递归代码一样，如果调用嵌套得太深可能会导致 Ruby 调用栈溢出。一些 Ruby 的实现会实验性地支持对尾调用的优化，这个技术能通过尽可能重用同样的栈帧来减少溢出风险。在 Ruby 的官方实现（MRI）里，我们可以这样打开尾调用优化：

```
RubyVM::InstructionSequence.compile_option = {
  tailcall_optimization: true,
  trace_instruction: false
}
```

为了确认生效，可以尝试对同样的 «while» 语句求值，这是之前用来检查小步语义的：

```
>> statement =
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     )
=> «while (x < 5) { x = x * 3 }»
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
```

这与小步语义给出的结果一致，所以看起来 While#evaluate 做的事情没错。

3. 应用

我们稍早时候对小步语义的实现只是适度使用了 Ruby 调用栈：在对一个大型程序调用

注 14：当然，没有什么能够阻止 Simple 程序员写出条件永远也不会为《false》的《while》语句，但如果那就是他们想要的，那也是可行的。

#reduce 时，消息会遍历抽象树直到其到达一段准备好规约的代码，这会引起一系列对 #reduce 的嵌套调用。[15] 但是伴随着反复执行小步规约，虚拟机通过维护当前程序和环境完成了对整个计算过程的跟踪；值得一提的是，嵌套调用只是用来遍历语法树查找下一步的规约对象，而不是执行规约本身，因此调用栈的深度受到程序语法树深度的限制。

相比之下，大步方式的实现会执行较小规模的计算，并将其作为更大规模计算的一部分。为了跟踪还有多少求值工作要做，它使用了更多的栈，并完全依赖栈来记住当前处理在整个计算中的位置。看上去像是对 #evaluate 的一次调用，实际上转换成了一系列递归调用，每一次调用都对一个子程序求值，这都让其在语法树中更进一步。

这个差别突出了每一种方法的目的。小步语义设定了一台能执行小操作的简单抽象机器，因此它包含了关于如何产生有用中间结果的详尽细节；大步语义把汇编整个计算的重担交给了机器或者执行它的人，在仅通过一步操作就把整个程序转换成一个最终结果的过程中，要求它跟踪许多中间子目标。根据我们想用一个语言的操作语义干什么——或是构建一个高效的实现，证明程序的某些属性，或是设计某个最佳变换——可能采用其中一种方法或者另一种方法会更合适。

大步语义在定义真正程序设计语言上最有影响的应用是第 6 章提到的标准 ML 编程语言（http://www.lfcs.inf.ed.ac.uk/reports/87/ECS-LFCS-87-36/）的原始定义，它用大步方式定义了 ML 的所有运行时行为。在这个例子之后，OCam 的核心语言用大步语义（http://caml.inria.fr/pub/docs/u3-ocaml/ocaml-ml.html#htoc7）补足了它更细节的小步定义。

W3C 也用到了大步操作语义：XQuery 1.0 和 XPath 2.0 规范（http://www.w3.org/TR/xquery-semantics/）使用数学化的推理规则描述它的语言应该如何求值，并且 XQuery 和 XPath 规范全文的 3.0 版本（http://www.w3.org/TR/xpath-full-text-30/）包括了一个使用 XQuery 写成的大步语义。

你可能注意到了，通过使用 Ruby 语言而不是数学语言写下 Simple 的小步和大步语义，我们已经为它实现了两个不同的 Ruby 解释器。操作语义实质上是这样的：通过描述一个解析器来说明一种语言的含义。正常情况下，这个描述应该用简单的数学符号来写，只要我们能理解，这将使一切都清晰而且无歧义，但是这样过于抽象而且离现实中的计算机有一定距离。把一种真实世界编程语言的额外复杂性（类、对象、方法调用……）引入到本该简约的说明当中，这是 Ruby 语言的缺点，但是如果我们已经理解 Ruby，那么就更容易理解整个过程，并且能够执行的描述可以当作一个解释器，这是个很好的红利。

注 15：有一种操作语义的替换形式，叫作规约语义，它通过引入所谓的规约上下文，把“下一步规约什么”和“如何对其进行规约”分离开来。这些上下文只是一些简明描述了规约在程序中何处发生的模式。这意味着我们只需要写真正执行计算的规约规则，从而把一些样板文件（boilerplate）从更大型的语言中去掉。

2.4 指称语义

到目前为止，我们已经从操作性方面观察了程序设计语言的含义，它通过展示程序执行之后发生的事情解释了程序的含义。而*指称语义*（denotational semantic）转而关心从程序本来的语言到其他表示的转换。

这种类型的语义没有直接处理程序的执行，而是关注如何借助另一种语言的已有含义——一种低级的、更形式化的或者至少比正在描述的语言更好理解的语言——解释一个新的语言。

指称语义确实是一种比操作语义更抽象的方法，因为它只是用一种语言替换另一种语言，而不是把一种语言转换成真实的行为。例如，如果我们需要向一个人解释英语动词“walk”的含义，但和他没有共同的口头语言，可以通过来回走的动作来沟通。另一方面，如果我们需要向一个说法语的人解释“walk”，可以跟他讲“marcher”——不可否认这是一种更高层次的沟通方式，不需要麻烦地运动了。

指称语义通常用来把程序转成数学化的对象，所以不出意料，可以用数学工具研究和控制它们，但是我们可以看看如何用另一种方式表示 Simple 程序，借此大致了解指称语义。

把 Simple 转成 Ruby 从而得到 Simple 语言的指称语义，[16] 事实上，这意味着把一个抽象语法树转成一个 Ruby 代码的字符串。不管怎样，我们得到了那种语法本来的含义。

但“本来的含义”是什么呢？我们表达式和语句的 Ruby 指称（denotation）是什么样的呢？从操作上我们已经看到一个表达式使用一个环境（environment）然后把它转成一个值；在 Ruby 中表达这个过程的一种方式是用一些参数表示环境参数，然后返回一些表示值的 Ruby 对象。对于像 «5» 和 «false» 这样简单的常量表达式，我们根本无需使用环境，而只需要关心它们最终的结果如何能表示成一个 Ruby 对象。幸运的是，Ruby 已经设计了专门的对象表示这些值：我们可以使用 Ruby 值 5 作为 Simple 表达式 «5» 的结果，同样地，把 Ruby 的值 false 作为 «false» 的结果。

2.4.1 表达式

我们可以用这个思想为 Number 类和 Boolean 类写一个 #to_ruby 的实现：

```
class Number
  def to_ruby
      "-> e { #{value.inspect} }"
  end
end
```

注 16：这意味着我们将用 Ruby 代码生成 Ruby 代码，但是选择用同样的指称语言和实现元语言只是为了让事情简单。例如我们很容易用 Ruby 写出能生成包含 JavaScript 字符串的代码来。

```
class Boolean
  def to_ruby
    "-> e { #{value.inspect} }"
  end
end
```

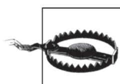
下面在控制台运行它们：

```
>> Number.new(5).to_ruby
=> "-> e { 5 }"
>> Boolean.new(false).to_ruby
=> "-> e { false }"
```

这些方法每个都产生一个刚好包含 Ruby 代码的字符串，并且因为 Ruby 是一种我们已经理解其含义的语言，所以可以看到这些字符串都是构造 proc 的程序。每一个 proc 都带有一个叫 `e` 的环境参数，它们完全忽略这个参数而直接返回一个 Ruby 值。

因为这些符号都是 Ruby 代码组成的字符串，所以可以使用 `Kernel#eval` 转换成可调用的 `Proc` 对象实际执行，然后在 IRB 中检查它们的行为 [17]：

```
>> proc = eval(Number.new(5).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> 5
>> proc = eval(Boolean.new(false).to_ruby)
=> #<Proc (lambda)>
>> proc.call({})
=> false
```



现阶段，完全避免 proc，而使用更简单的 `#to_ruby` 实现是很诱人的，这只需要把 `Number.new(5)` 转换成字符串 `'5'` 而不是 `'-> e {5}'` 等，但是从源语言结构中获得其本质语义是指称语义这一方法的一部分，那么我们需要知道，即便某些特定的表达式不会用到环境，通常的表达式也还是需要一个环境的。

为了表示确实使用环境的表达式，我们需要决定如何用 Ruby 表示环境（environment）。在研究操作语义时我们已经了解了环境，那么既然它们已经用 Ruby 实现了，现在可以重用早期的思想——把一个环境表示成一个散列表。不过细节需要做一些改动，因此要注意其中微妙的差别：在我们的操作语义中，环境是生存在虚拟机中的，并且把变量名与 `Number.new(5)` 这样的 Simple 抽象语法树联系起来；但在我们的指称语义中，环境存在于我们要把程序转换得到的语言中，因此要在那个世界而不是在一个虚拟机的“外部世界”起作用。

注 17：只有 Ruby 既做实现语言又作为指称语言的时候我们才能这么做。如果指称是 JavaScript 源代码，我们就得到 JavaScript 的控制台去实验它们了。

注意，这意味着指称环境（denotational environment）应该把变量名与 5 这样的原生 Ruby 值，而不是与表示 Simple 语法的对象关联起来。我们把 { x: Number.new(5) } 这样的操作环境（operational environment）看成在要转换成的语言中拥有指称 '{ x: 5 }'，并且因为实现的元语言和指称语言正好都是 Ruby，所以不必有什么顾忌。

既然知道环境将是一个散列，那么就可以实现 Variable#to_ruby 了：

```
class Variable
  def to_ruby
    "-> e { e[#{name.inspect}] }"
  end
end
```

这段代码，把一个变量表达式转换成一个在环境散列中查找合适值的 Ruby proc：

```
>> expression = Variable.new(:x)
=> «x»
>> expression.to_ruby
=> "-> e { e[:x] }"
>> proc = eval(expression.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 7 })
=> 7
```

关于指称语义重要的一点是它是*组合式*的：一个程序的指称由组成它的各部分的指示构成。在开始指称（denotating）Add、Multiply 和 LessThan 这样的更大表达式时，我们就能理解这种合成性了：

```
class Add
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) + (#{right.to_ruby}).call(e) }"
  end
end

class Multiply
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) * (#{right.to_ruby}).call(e) }"
  end
end

class LessThan
  def to_ruby
    "-> e { (#{left.to_ruby}).call(e) < (#{right.to_ruby}).call(e) }"
  end
end
```

这里使用字符串串联操作把子表达式的指称组成一个大表达式的指称。我们知道每一个子表达式都将在 Ruby 源码中用一个 proc 表示，因此可以将它们作为更大段 Ruby 代码的一部分，那些更大段的代码使用提供的环境调用这些 proc，并使用它们返回的值进行一些计

算。下面是得到结果：

```
>> Add.new(Variable.new(:x), Number.new(1)).to_ruby
=> "-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }"
>> LessThan.new(Add.new(Variable.new(:x), Number.new(1)), Number.new(3)).to_ruby
=> "-> e { (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) }).call(e) < (-> e {
3 }).call(e) }"
```

这些指称已经够复杂的了，很难了解它们做的事情是否正确。让我们运行它们确认一下：

```
>> environment = { x: 3 }
=> {:x=>3}
>> proc = eval(Add.new(Variable.new(:x), Number.new(1)).to_ruby)
=> #<Proc (lambda)>
>> proc.call(environment)
=> 4
>> proc = eval(
     LessThan.new(Add.new(Variable.new(:x), Number.new(1)), Number.new(3)).to_ruby
   )
=> #<Proc (lambda)>
>> proc.call(environment)
=> false
```

2.4.2　语句

我们可以用类似的方式定义语句的指称语义，但是要记住操作语义中提到的：对一个语句求值产生的是一个新的环境而不是一个值。这意味着 `Assign#to_ruby` 需要为 proc 构造一些代码，以使结果是一个更新了的环境散列：

```
class Assign
  def to_ruby
    "-> e { e.merge({ #{name.inspect} => (#{expression.to_ruby}).call(e) }) }"
  end
end
```

还是可以在控制台对其进行检查：

```
>> statement = Assign.new(:y, Add.new(Variable.new(:x), Number.new(1)))
=> «y = x + 1»
>> statement.to_ruby
=> "-> e { e.merge({ :y => (-> e { (-> e { e[:x] }).call(e) + (-> e { 1 }).call(e) })
.call(e) }) }"
>> proc = eval(statement.to_ruby)
=> #<Proc (lambda)>
>> proc.call({ x: 3 })
=> {:x=>3, :y=>4}
```

和之前一样，`DoNothing` 的语义非常简单：

```
class DoNothing
  def to_ruby
```

```
    '-> e { e }'
  end
end
```

对于条件语句，我们可以把 Simple 的 «if (...) { ... } else { ... }» 转换成一个 Ruby 的 if ... then ... else ... end，确保环境传到了需要它的地方：

```
class If
  def to_ruby
    "-> e { if (#{condition.to_ruby}).call(e)" +
      " then (#{consequence.to_ruby}).call(e)" +
      " else (#{alternative.to_ruby}).call(e)" +
      " end }"
  end
end
```

就像在大步操作语义中一样，我们需要小心地定义序列语句：对第一个语句求值的结果作为对第二个语句求值时的环境。

```
class Sequence
  def to_ruby
    "-> e { (#{second.to_ruby}).call((#{first.to_ruby}).call(e)) }"
  end
end
```

最后，就像处理条件语句那样，我们可以把 «while» 语句转成 proc，在返回最终环境之前，它使用 Ruby 的 while 重复执行语句主体：

```
class While
  def to_ruby
    "-> e {" +
      " while (#{condition.to_ruby}).call(e); e = (#{body.to_ruby}).call(e); end;" +
      " e" +
      " }"
  end
end
```

哪怕是一个简单的 «while» 都具有一个冗长的表示，所以有必要用 Ruby 解释器检查一下它的含义正确与否：

```
>> statement =
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Multiply.new(Variable.new(:x), Number.new(3)))
     )
=> «while (x < 5) { x = x * 3 }»
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e) }).call(e);
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) * (-> e { 3 }).call(e)
}).call(e) }) }).call(e); end; e }"
>> proc = eval(statement.to_ruby)
```

```
=> #<Proc (lambda)>
>> proc.call({ x: 1 })
=> {:x=>9}
```

语义类型比较

«while» 是一个区分小步语义、大步语义和指称语义的好例子。

«while» 的小步操作语义是以一台抽象机器的归约规则形式写成的。整个循环并不是规约行为的一部分——规约只是把一个 «while» 语句转成一个 «if» 语句——但是它会作为将来由机器执行的规约序列的一部分。为了理解 «while» 做了什么，我们需要考虑所有的小步规则，并弄懂随着一个 Simple 程序的执行它们之间是如何互相作用的。

«while» 的大步操作语义是以一个求值规则的形式写成的，这个规则说明如何把最终的环境直接计算出来。这个规则包含了对其本身的递归调用，因此明显表明 «while» 在求值过程中会引发一个循环，但不是 Simple 程序员熟悉的那种循环。大步的规则是递归的形式，描述了如何根据对其他语法结构的求值对一个表达式或者语句完整地求值，因此这个规则告诉我们，对一个 «while» 语句求值的结果可能会依赖于一个不同环境下同样语句的求值结果，但把这种思想与 «while» 应该展现的迭代方式联系起来需要跳跃性思维。幸运的是这种跳跃并不太大：一点点的数学推理可以表明两种类型的循环在本质上是等价的，并且在元语言支持尾调用优化的时候，它们事实上也是等价的。

«while» 的指称语义展示了如何用 Ruby 对其重写，也就是如何通过 Ruby 的 while 关键字对其重写。这是一个简单直接得多的转换：Ruby 提供对迭代循环的原生支持，而指称规则也表明 «while» 能用 Ruby 的这个特性实现。要理解这两种类型的循环没有什么困难，所以如果我们理解了 Ruby 中 while 循环的工作方式，也能理解 Simple 的 «while» 循环。当然，这意味着我们已经把理解 Simple 的问题转换成了理解指称语言的问题，而如果指称语言像 Ruby 一样庞大而且定义不良，这就是一个严重的缺点；但在有一个能用来写指称的小型数学语言时，这就成了一个优点。

2.4.3 应用

做完所有这些工作之后，指称语义完成了什么目标呢？它的主要目的是展示如何把 Simple 翻译成 Ruby，它将后者作为工具来解释不同的语言结构是什么意思。这恰巧给了我们执行 Simple 程序的一种途径——因为已经用可执行的 Ruby 写下了指称语义的规则，而且这些规则的输出本身就是可执行的 Ruby——但这只是偶然事件，因为我们之前有可能用普通的英语写规则并用一些数学语言写下指称。真正重要的是我们自己随意设计了一种语言，并把它转换成一种其他人或者其他东西能理解的语言。

为了赋予这种转换一些解释能力，把一部分语言含义放到表面而不再只是隐含在背后会非常有帮助。例如，这种语义把环境表示成具体的 Ruby 对象——在 proc 中传入和返回的散列，而不是把 Simple 中的变量表示成真正的 Ruby 变量，然后依赖 Ruby 自己微妙的变量作

用域规则去定义 Simple 的变量访问机制；这样表示环境更为明确直接。在这方面这种语义除了把解释性的工作交给 Ruby，还多做了一些事情；它把 Ruby 作为一个简单的基础，但是在表面做了一些额外的工作，从而准确地展示了不同程序结构是如何使用和改变环境的。

这之前我们看到过，操作语义通过为一种语言设计一个解释器来解释这种语言的含义。与此对比，语言到语言的指称语义更像是一个*编译器*：在这种情况下，我们的 `#to_ruby` 实现高效地把 Simple 编译成 Ruby。这些类型的语义虽然都对如何为一种语言*高效地*实现一个解释器或者编译器只字不提，但确实提供了一个基础标准可以检验任何生效了的实现。

这些指称的定义还在一些语言的原始状态中出现过。早期版本的 Scheme 标准使用指称语义（http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%25_sec_7.2）定义核心语言，而不像现在的标准使用小步操作语义来定义，并且 XSLT 文本转换语言的开发是由 Philip Wadler 对 XSLT 模式（http://homepages.inf/ed.ac.uk/wadler/topics/xml.html#xsl-semantics）和 XPath 表达式（http://homepages.inf.ed.ac.uk/wadler/topics/xml.html#xpath-semantics）的指称定义来引导的。

3.3.2 节有一个实际使用指称语义定义正则表达式的例子。

2.5 形式化语义实践

对于为计算机程序赋予含义的问题，本章已经展示了几种不同的方法。在每种情况下，我们都已经避免了数学化的方法并使用 Ruby 了解了它们的策略，但是形式化的语义通常都是由数学化的工具完成的。

2.5.1 形式化

我们对形式语义的研究并不是特别正式。一直没有认真关注过数学符号，而使用 Ruby 作为元语言意味着比起理解程序的各种方式，我们更关注执行程序的不同方式。合适的指称语义关注的是通过把程序转换成定义良好的数学对象以获得程序的核心含义，关心的是把一个 Simple 的 «`while`» 语句无歧义的完整表示成一个 Ruby 的 `while` 循环。



为了提供对指称语义有用的定义和对象，专门发展了称为*域理论*的数学分支，它采用基于单调函数上不动点的一种计算模型，并且这个单调函数定义在偏序集合上。我们可以通过把程序“编译”成数学函数来理解这个程序，并且域理论的技巧还能用来证明这些函数一些有趣的特性。

另一方面，尽管我们只是用 Ruby 含糊地概括了一下指称语义，但关于操作语义，我们已经在精神上接近它的形式化表示了：我们对方法 `#reduce` 和 `#evaluate` 的定义实际上只是用 Ruby 翻译的数学化推理规则。

2.5.2 找到含义

形式化语义的一个重要应用是为一种编程语言的含义给出一个无歧义的定义，而不是让其依赖于像自然语言规范文档和“由实现规范”这样更加随意的方法。形式化的定义还有其他用途，例如证明某种语言通常情况下的特性，以及特定程序在特定情况下的特性，证明语言中程序之间的等价性，研究如何在不改变程序行为的情况下安全地变换程序而使其效率更高。

例如，既然操作语义与解释器的实现极为接近，那么计算机科学家就可以把一个适当的解释器看成一种语言的操作语义，然后证明它在那种语言的指称语义方面的正确性——这意味着证明了由解释器给出的含义和由指称语义给出的含义之间存在着明显的联系。

指称语义的一个优点是比操作语义抽象层次更高，它忽略了程序如何执行的细节，而只关心如何把它转换成一个不同的表示。例如，如果存在一种指称语义可以把两种语言翻译成某种共通的表示，就使对不同语言写成的两个程序进行比较成为可能。

抽象程度会使指称语义看起来有点兜圈子。如果问题是如何解释一种程序设计语言的含义，那么把一种语言翻译成另一种语言是如何让我们更接近问题答案的呢？一个指称只不过与它的含义一样好；尤其是，如果指称的语言有某种*操作性*的含义，那么一个指称语义只是让我们更接近于能实际执行一个程序，这个语言的语义本身展示了它是如何执行的，而不是如何翻译成另一种语言的。

形式化的指称语义使用抽象的数学对象（通常是函数）来表示表达式和语句这样的编程语言结构，并且因为数学上的约定会规定如何对函数求值这样的事情，这就有了一种直接在操作意义上思考指称的方式。我们已经使用了不太正式的方式，把指称语义看成是一种语言到另一种语言的编译器，而事实上这是多数编程语言最终得以执行的方式：一个 Java 程序将会由 javac 编译成字节码，字节码将会被 java 的虚拟机即时编译成 x86 的指令，然后一个 CPU 会把每一条 x86 指令解码成类 RISC（精简指令集）的微指令放到一个核上去执行……它会在什么地方结束呢？是编译器，还是虚拟机，还是一直重复下去？

当然程序最终会执行，因为语义这个高楼会到达底部暴露出*实际*的机器：半导体中的电子，它们遵守的是物理法则。[18] 一台计算机是维护这个不确定结构的装置，大量复杂的解释层在彼此之上保持稳定平衡，这就允许多点触控手势这样人体尺度的想法和 `while` 循环这样的想法，都能被逐渐地向下翻译给硅和电的物理世界。

2.5.3 备选方案

本章你已经看到了许多不同名称的语义类型。小步语义还叫*结构化操作语义*（structural

注 18：或者，在 Charles Babbage 设计的分析机这种机械计算机的场景下，是齿轮和纸遵守物理规律。

operational semantic）和转换语义（transition semantic）；大步语义更普遍的叫法是自然语义（natural semantic）或者关联语义（relational semantic）；而指称语义还可以称为不动点语义（fixed-point semantic）或者数学语义（mathematical semantic）。

还有其他类型的形式语义可用。其中一个就是公理化语义（axiomatic semantic），它通过在语句执行前后分别给出抽象机器状态的断言来描述一个语句的含义：如果一个断言（前置条件）在语句执行前初始是 true，那么随后的其他断言（后置条件）将是 true。公理化语义在验证程序的正确性方面很有用：随着语句合到一起组成更大的程序，它们对应的断言也能合到一起组成更大的断言，其目标就是表明对一个程序总体的断言与它的预期定义匹配。

虽然细节有所不同，但是公理化语义是描述 RubySpec project 最好的语义类型，RubySpec project（http://www.rubyspec.org）是“Ruby 程序设计语言的可执行规范”，它使用 RSpec 类型的断言既描述 Ruby 的核心以及标准库，又描述 Ruby 内置语言结构的行为。例如，下面是 RubySpec 描述 Array#<< 方法的片段：

```
describe "Array#<<" do
  it "correctly resizes the Array" do
    a = []
    a.size.should == 0
    a << :foo
    a.size.should == 1
    a << :bar << :baz
    a.size.should == 3

    a = [1, 2, 3]
    a.shift
    a.shift
    a.shift
    a << :foo
    a.should == [:foo]
  end
end
```

2.6 实现语法解析器

本章，我们已经手工构建了 Simple 程序的抽象语法树——通过手写 `Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))` 这样的普通 Ruby 表达式，而不是先写 `'x = x + 1'` 这样原始的 Simple 源代码，然后使用一个语法解析器自动地把它转成语法树。

从头开始完整地实现一个 Simple 的语法解析器过于复杂，会分散我们讨论形式语义的注意力。尽管破解一个小编程语言很有趣，但是感谢解析工具和解析库的存在，在他人工作的基础上构造一个语法解析器并不是特别困难，因此下面将对其简单介绍一下。

Treetop（http://treetop.rubyforge.org/）是 Ruby 可用的语法解析工具中最好的一个，它是一

种特定领域的语言，能让语法解析器自动生成。一种语言的 Treetop 描述会写成*解析表达式语法*（parsing expression grammar），这是一个简单的类正则表达式（regular-expression-like ）的规则集合，既易写又易理解。最好的是，这些规则能够使用方法定义作为注释，这样的话，就可以为语法解析过程中生成的 Ruby 对象定义行为。Treetop 既能定义语法结构，又能定义基于这些结构进行运算的 Ruby 代码集合，这使 Treetop 很适合描述一种语言的语法并赋予它可执行的语义。

为了让我们体验一下这是如何工作的，下面给出关于 Simple 的 Treetop 语法简装版，它只包含解析字符串“`while (x < 5) { x =x * 3 }`”所需要的规则：

```
grammar Simple
  rule statement
      while / assign
  end

  rule while
    'while (' condition:expression ') { ' body:statement ' }' {
      def to_ast
        While.new(condition.to_ast, body.to_ast)
      end
    }
  end

  rule assign
    name:[a-z]+ ' = ' expression {
      def to_ast
        Assign.new(name.text_value.to_sym, expression.to_ast)
      end
    }
  end

  rule expression
    less_than
  end

  rule less_than
    left:multiply ' < ' right:less_than {
      def to_ast
        LessThan.new(left.to_ast, right.to_ast)
      end
    }
    /
    multiply
  end

  rule multiply
    left:term ' * ' right:multiply {
      def to_ast
        Multiply.new(left.to_ast, right.to_ast)
      end
    }
    /
    term
```

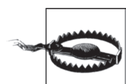
```
  end

  rule term
    number / variable
  end

  rule number
    [0-9]+ {
      def to_ast
        Number.new(text_value.to_i)
      end
    }
  end
  rule variable
    [a-z]+ {
      def to_ast
        Variable.new(text_value.to_sym)
      end
    }
  end
end
```

这种语言看起来有点像 Ruby，但这种相似性只是表面的；语法是用特别的 Treetop 语言写出来的。关键字 `rule` 为分析一种特定种类的语法引入一个新的规则，并且每个规则里的表达式描述了它将要识别的字符串结构。规则可以递归地调用其他规则——例如 `while` 规则调用表达式（expression）规则和语句（statement）规则——而且分析从第一条规则开始，这是这种语法中的语句。

这些表达式语法规则彼此调用的顺序反应了 Simple 运算符的优先级。表达式语法调用 `less_than`，然后 `less_than` 立即调用 `multiply`，在 `less_than` 对优先级更低的运算符 `<` 进行匹配之前，`multiply` 能在字符串中匹配到 `*` 运算符。这确保表达式 `'1 * 2 < 3'` 被解析成 «`(1 * 2) < 3`» 而不是 «`1 * (2 < 3)`»。



为了让事情简单，这个语法没有试图限制可以在一种表达式中出现的另一种表达式种类，这意味着这个表达式将会接受一些明显错误的程序。

例如，对于二元表达式 `less_than` 和 `multiply`，我们设定了两个规则——但是分别设立两个规则的唯一原因是为了强调运算符的优先级，这样每一个规则只要求一个更高优先级的规则匹配其左侧运算对象，然后同样或者更高优先级的规则匹配其右侧运算对象。这将使像 `'1 < 2 < 3'` 这样的字符串能成功通过解析，即便 Simple 的语义无法赋予这个表达式结果一个含义。

这些问题中有一些可以通过对语法稍作调整得以解决，但是总会有其他一些不正确的情况语法解析器不能识别。这一问题我们将分成两个关注点，首先保持语法解析器尽可能的自由，其次将在第 9 章使用一个不同的技术来检测无效的程序。

语法中大多数的规则都使用外边带上括号的 Ruby 代码标注。在每一个括号里，代码都定义一个叫 #to_ast 的方法，在解析一个 Simple 程序的时候，它能用在由 Treetop 构建的对应语法对象上。

如果把这个语法保存到叫作 simple.treetop 的文件里，我们可以使用 Treetop 加载它来生成一个 SimpleParser 类。这个解析器可以把一个由 Simple 源代码组成的字符串转换成由 Treetop 的 SyntaxNode 对象构建出来的一个表示：

```
>> require 'treetop'
=> true
>> Treetop.load('simple')
=> SimpleParser
>> parse_tree = SimpleParser.new.parse('while (x < 5) { x = x * 3 }')
=> SyntaxNode+While1+While0 offset=0, "...5) { x = x * 3 }" (to_ast,condition,body):
     SyntaxNode offset=0, "while ("
     SyntaxNode+LessThan1+LessThan0 offset=7, "x < 5" (to_ast,left,right):
       SyntaxNode+Variable0 offset=7, "x" (to_ast):
         SyntaxNode offset=7, "x"
       SyntaxNode offset=8, " < "
       SyntaxNode+Number0 offset=11, "5" (to_ast):
         SyntaxNode offset=11, "5"
     SyntaxNode offset=12, ") { "
     SyntaxNode+Assign1+Assign0 offset=16, "x = x * 3" (to_ast,name,expression):
       SyntaxNode offset=16, "x":
         SyntaxNode offset=16, "x"
       SyntaxNode offset=17, " = "
       SyntaxNode+Multiply1+Multiply0 offset=20, "x * 3" (to_ast,left,right):
         SyntaxNode+Variable0 offset=20, "x" (to_ast):
           SyntaxNode offset=20, "x"
         SyntaxNode offset=21, " * "
         SyntaxNode+Number0 offset=24, "3" (to_ast):
           SyntaxNode offset=24, "3"
     SyntaxNode offset=25, " }"
```

这个 SyntaxNode 结构是一个*具体语法树*：它专门为了 Treetop 的处理而设计，并且含有关于这个具体语法树的节点是如何与生成它们的原始代码关联起来的大量信息。下面是 Treetop 文档（http://treetop.rubyforge.org/using_in_ruby_html）不得不说的一些话：

> 请不要尝试自己向下遍历语法树，并且不要把这棵树的结构作为你自己常用的数据结构。它包含的节点比你应用程序所需要的要多得多，甚至为输入的每个字符都分配一个还绰绰有余。
>
> 但是，你可以为根规则增加方法，根规则以一种合理的格式返回你需要的信息。每个规则可以调用它的子规则，并且从外面尝试遍历树时，利用这些遍历语法树的方法是一个非常好的选择。

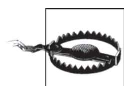
这就是我们已经做到的。我们没有直接操纵这棵乱糟糟的树，而是使用语法中的标记在每个节点上定义一个 #to_ast 方法。如果在根节点上调用这个方法，它会根据 Simple 的语法

对象构建一棵抽象语法树。

```
>> statement = parse_tree.to_ast
=> «while (x < 5) { x = x * 3 }»
```

这样我们已经自动地把源代码转换成了一棵抽象语法树，并且现在可以使用这棵树以通常的方式查看程序的含义了：

```
>> statement.evaluate({ x: Number.new(1) })
=> {:x=>«9»}
>> statement.to_ruby
=> "-> e { while (-> e { (-> e { e[:x] }).call(e) < (-> e { 5 }).call(e) }).call(e);
e = (-> e { e.merge({ :x => (-> e { (-> e { e[:x] }).call(e) * (-> e { 3 }).call(e)
}).call(e) }) }).call(e); end; e }"
```



这个解析器和 Treetop 通常还有一个缺点，就是生成一个*右结合*的具体语法树。这意味着字符串 `'1 * 2 * 3 * 4'` 被解析时会被当成：`'1 * (2 * (3 * 4))'`：

```
>> expression = SimpleParser.new.parse('1 * 2 * 3 * 4', root: :expression).to_ast
=> «1 * 2 * 3 * 4»
>> expression.left
=> «1»
>> expression.right
=> «2 * 3 * 4»
```

但是乘法通常是*左结合*的：写 `'1 * 2 * 3 * 4'` 的时候，我们实际的意思是 `'((1 * 2) * 3) * 4'`，这里数字是从表达式的左边（而非右边）开始分组结合的。对乘法来说这没什么关系——求值的时候两种方式会产生同样的结果——但对像减法和除法这样的运算就有问题了，因为对 `«((1 - 2) - 3) - 4»` 求值的结果与对 `«1 - (2 - (3 - 4))»` 求值的结果*并不相同*。

为了修正这个缺点，我们不得不让这些规则和 `#to_ast` 实现得更加复杂一些。参考 6.2.3 节，那里有构建左结合 AST 的 Treetop 语法。

能够像这样解析 Simple 程序很方便，但是因为困难的工作都由 Treetop 做了，所以我们对一个语法解析器实际如何工作并没有了解多少。在 4.3 节，你将会看到如何直接地实现一个解析器。

第 3 章

最简单的计算机

短短的几年里，我们已经身处计算机的海洋。本来它们都安全地隐藏在军事研究中心和大学实验室中，但现在已经随处可见：我们的办公桌上，我们的口袋里，汽车的发动机罩下，甚至植入了我们的身体。作为程序员，我们每天都在使用精密的计算机，但对它们的工作方式了解多少呢？

现代计算机的强大能力伴随着过多的复杂性。我们很难理解一台计算机多个子系统的全部细节，更别说理解那些子系统如何互相协作从而构成整个系统了。这些复杂性使得对真实计算机的能力与行为进行直接推导显得不切实际，此时计算机的简化模型就显得很有用了，虽然模型只是提取出真实计算机中令人感兴趣的特性，但它确实能够帮助人们建立完整的认识。

本章，我们将抽丝剥茧，揭开计算机的本质，看看它到底能干些什么，并考察这样一台简单计算机所能完成工作的极限。

3.1 确定性有限自动机

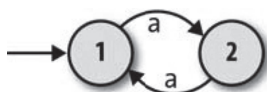
现实中，计算机通常都有大量的易失存储器（RAM）和非多核易失存储器（硬盘或者 SSD），有许多输入 / 输出设备，还有能同时执行多个指令的处理器。*有限状态机*（finite state machine），也叫*有限自动机*（finite automaton），是一台计算机的极简模型，为了容易理解、推导并且容易用硬件或软件实现，它放弃了上面所有的这些特性。

3.1.1 状态、规则和输入

有限自动机没有持久化的存储并且几乎没有 RAM。它只是一台小机器，拥有一些可能的*状态*，并能够跟踪到自己当前具体处于其中的哪个状态——试着把它看成一台 RAM 只够存储一个值的计算机。同样，有限自动机没有键盘、鼠标和接收输入的网络接口，只有一个外部的字符输入流可以一次读取一个字符。

每台有限自动机没有通用的 CPU 执行任意程序，而是硬编码了一些*规则*集合，以决定在相应的输入下如何从一个状态切换到另一个状态。自动机先从一个特定的状态开始，然后从输入流中读入字符——按照规则它每次读取一个字符。

下面是一台有限自动机的结构图：



两个圆代表自动机的两个状态——1 和 2。凭空出现的箭头表明这台自动机从状态 1 开始，1 是它的*起始状态*。两个状态之间的箭头代表机器的规则：

- 处于状态 1 并且读入字符 a 时，切换到状态 2；
- 处于状态 2 并且读入字符 a 时，切换到状态 1。

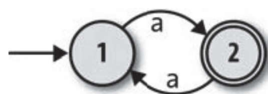
这让我们有足够的信息研究机器如何处理一个输入流。

- 这台机器从状态 1 开始。
- 这台机器只有从输入流读入字符 a 的规则，因此这是唯一能发生的事情。读取到 a 的时候，它会从状态 1 切换到状态 2。
- 当这台机器又读取到了一个 a 时，它会切换回状态 1。

一旦回到状态 1，它又将开始重复自身，这就是这台机器的行为范围。我们可以认为当前状态的信息存在于机器内部——它像一个“黑盒”一样运转，并不会展现其内部工作状况——这台无聊的机器毫无用处，没有任何能观察到的输出。即使这台机器一直在状态 1 和状态 2 之间切换，机器之外也没有一个人能看出来有什么事情在发生。因此在这种情况下，我们可能还要增加一个状态，这样就不用再为任何内部结构操心了。

3.1.2 输出

为了解决这个问题，有限自动机还有一个产生输出的基本方法。与现实中计算机复杂的输出能力相比这不值一提，我们只是把一些状态标记成特别状态，并且认为机器的单比特输出提供了当前是否处于特别状态的信息。对于这台机器，我们将状态 2 作为特别状态，并在图中用双重的圆形表示它。

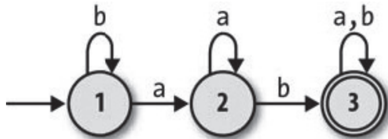


这些特定状态通常称为*接受状态*，表明这台机器对某个输入序列是*接受*还是*拒绝*。如果这台自动机从状态 1 开始并读入一个 a，它将会停留在状态 2，这是一个接受状态，因此我们可以说这台机器接受字符串 'a'。另外，如果它先读到一个 a，然后又读取了另一个 a，它将终止于状态 1，这不是一个接受状态，所以这台机器拒绝字符串 'aa'。事实上很容易看到，这台机器接受任何奇数个数的 a 组成的字符串：'a'、'aaa'、'aaaaa' 都能被接受，但是 'aa'、'aaaa' 和 ''（空字符串）会被拒绝。

现在有了稍有用一些的东西：一台机器，它能读取一个字符序列，并且提供一个“是 / 否”的输出，以表明这个序列是否已经被接受。公道地说，这个 DFA（Deterministic Finite Automata）正在执行计算，因为我们可以向它提问——“这个字符串的长度是奇数吗？”——然后得到一个有意义的答案。它足以称为简单计算机了，并且我们可以将它的特性与一台现实中的计算机进行对比：

	真实计算机	有限自动机
持久存储	硬盘或者 SSD	无
临时存储	RAM	当前状态
输入	键盘、鼠标、网络等	字符流
输出	显示设备、话筒、网络等	当前状态是否为一个接受状态（是 / 否）
处理器	能执行任何程序的 CPU 核心	根据输入改变状态的硬编码规则

当然，这台自动机不做任何精细或者有用的工作，但是我们可以构造更复杂的自动机，让它拥有更多的状态并且能够读取多个字符。下面的自动机有三个状态，并且能够读取输入 a 和 b：



这台机器接受 'ab'、'baba' 以及 'aaaab' 这样的字符串，并且拒绝 'a'、'baa' 和 'bbbba' 这样的字符串。实验表明，它只接受包含序列 'ab' 的字符串，因此仍然没有多大用，但至少展现了一定程度的精妙之处。本章后面我们将看到更实际的应用。

3.1.3 确定性

很明显，这种自动机具有*确定性*：不管它当前处于什么状态，并且不管读入什么字符，最终所处的状态总是完全确定的。只要满足下面两个约束，就能保证这种确定性。

- **没有冲突** 不存在这样的状态：它的下一次转换状态因为有彼此冲突的规则而有二义性。（这意味着一个状态对于同样的输入，不能有多个规则。）
- **没有遗漏** 不存在这样的状态：它的下一次转换状态因为缺失规则而未知。（这意味着每个状态都必须针对每个可能的输入字符有至少一个规则。）

综上所述，这些约束意味着对每一个状态和输入的组合，这台机器一定要恰好有一个规则。遵守这些确定性约束的机器有一个技术名称，就是*确定性有限自动机*（Deterministic Finite Automaton，DFA）。

3.1.4 模拟

确定性有限自动机是计算的抽象模型。我们已经画了一些示例机器的简图，而且思考了它们的行为，但是这些机器实际上并不存在，因此我们不能真正给它们一些输入然后看它们的表现。幸运的是，DFA 非常简单，我们很容易用 Ruby 对其进行模拟，然后直接与它交互。

让我们通过实现一个规则集合对其进行模拟，并把这个规则集合称为*规则手册*（rulebook）：

```
class FARule < Struct.new(:state, :character, :next_state)
  def applies_to?(state, character)
    self.state == state && self.character == character
  end

  def follow
    next_state
  end

  def inspect
    "#<FARule #{state.inspect} --#{character}--> #{next_state.inspect}>"
  end
end

class DFARulebook < Struct.new(:rules)
  def next_state(state, character)
    rule_for(state, character).follow
  end

  def rule_for(state, character)
    rules.detect { |rule| rule.applies_to?(state, character) }
  end
end
```

这段代码为规则建立了一个简单的 API：每个规则都有一个 `#applies_to?` 方法（这个方法会返回 `true` 或者 `false`，指示这个规则是否可以在某个特定情况下应用），还有一个 `#follow` 方法（在决定采用某条规则后返回关于机器应该如何改变的信息）。[1] `DFARulebook#next_state`

注 1：这个设计足够通用，可以适应不同种类的机器和规则，因此在本书稍后情况更复杂的情况下我们还可以重用它。

使用这些方法定位到正确的规则，并找到 DFA 接下来的状态。



通过使用 Enumerable#detect，DFARulebook#next_state 的实现假定总是恰好有一个规则应用到给定的状态和字符上。如果可用的规则超过一个，那么只有第一个能起作用，其他规则都会被忽略；如果没有可以应用的规则，#detect 调用会返回 nil，并且在试图调用 nil.follow 的时候模拟进程会崩溃。

这就是为什么这个类叫 DFARulebook 而不是 FARulebook 了：它只是在确定性约束满足的情况下才正确工作。

一个规则手册能够把许多规则封装到一个对象里，然后询问它接下来是什么状态：

```
>> rulebook = DFARulebook.new([
     FARule.new(1, 'a', 2), FARule.new(1, 'b', 1),
     FARule.new(2, 'a', 2), FARule.new(2, 'b', 3),
     FARule.new(3, 'a', 3), FARule.new(3, 'b', 3)
   ])
=> #<struct DFARulebook ...>
>> rulebook.next_state(1, 'a')
=> 2
>> rulebook.next_state(1, 'b')
=> 1
>> rulebook.next_state(2, 'b')
=> 3
```



此处我们面临一个选择，即如何把自动机的状态表示成 Ruby 的值。重点在于能把这些状态区分开来：我们对 DFARulebook#next_state 的实现需要能够比较两个状态，以判定它们是否相同，但并不关心那些对象是数字、符号、字符串、散列，还是 Object 类的匿名实例。

在这种情况下，最清晰的方式是使用普通的 Ruby 数字——它们能很好地匹配图中带编号的状态，因此我们就是这么做的。

有了一个规则手册之后，我们可以用它来构建一个 DFA 对象，以跟踪它的当前状态，并且可以报告它当前是否处于接受状态：

```
class DFA < Struct.new(:current_state, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_state)
  end
end

>> DFA.new(1, [1, 3], rulebook).accepting?
=> true
>> DFA.new(1, [3], rulebook).accepting?
=> false
```

现在可以写一个方法从输入中读取一个字符，然后查阅规则手册，再相应地改变状态：

```
class DFA
  def read_character(character)
    self.current_state = rulebook.next_state(current_state, character)
  end
end
```

为 DFA 输入字符串，然后观察它输出的改变：

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> false
>> 3.times do dfa.read_character('a') end; dfa.accepting?
=> false
>> dfa.read_character('b'); dfa.accepting?
=> true
```

一次只向 DFA 输入一个字符有些不方便，所以添加一个方便的方法来读取输入的整个字符串：

```
class DFA
  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

现在可以向 DFA 输入整个字符串了，而不再只是分别传入单个字符：

```
>> dfa = DFA.new(1, [3], rulebook); dfa.accepting?
=> false
>> dfa.read_string('baaab'); dfa.accepting?
=> true
```

一旦 DFA 获得了一些输入，它就可能不再处于起始状态了，因此我们不能再次使用它检查输入的一个新的完整序列。这意味着要从头创建它——像以前那样使用同样的起始状态、接受状态和规则手册——每当想要检查它是否接受一个新的字符串时。我们可以在一个对象里封装它的构造参数来避免手工执行这一操作，这个对象表示设计出来的特定 DFA，只要我们想要检查是否可以接受一个新的字符串，就靠此对象自动地构建那个 DFA 的一次性实例：

```
class DFADesign < Struct.new(:start_state, :accept_states, :rulebook)
  def to_dfa
    DFA.new(start_state, accept_states, rulebook)
  end

  def accepts?(string)
```

```
    to_dfa.tap { |dfa| dfa.read_string(string) }.accepting?
  end
end
```



#tap 方法对一个代码块求值，然后返回调用它的对象。

DFADesign#accepts? 使用 DFADesign#to_dfa 方法创建一个 DFA 的新实例，然后调用 #read_string? 把它放到一个接受态或者拒绝态里：

```
>> dfa_design = DFADesign.new(1, [3], rulebook)
=> #<struct DFADesign ...>
>> dfa_design.accepts?('a')
=> false
>> dfa_design.accepts?('baa')
=> false
>> dfa_design.accepts?('baba')
=> true
```

3.2 非确定性有限自动机

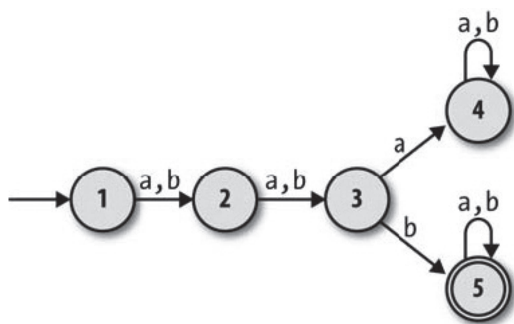
DFA 理解和实现起来都很简单，但那是因为它与我们熟悉的机器非常相似。在去除一台真实计算机的所有复杂性之后，我们有机会使用不太常见的思想进行实验了，这将让我们远离熟悉的机器，并可以不必处理把这些思想落实到真实系统中时可能遇到的各种困难。

一种探索方式是去掉我们现有的假设和约束。首先，确定性约束似乎是个限制：可能我们并不关心每个状态上每个可能的输入，那么为什么不能忽略不关心的字符处理规则，而假设异常发生时这台机器能进入到一个通用的失败状态呢？更异乎寻常的是，如果允许这台机器拥有互相对立的规则，以致有多条可能的执行路径，这将意味着什么呢？我们之前的设置还假设，每一个状态改变一定对应从输入流读入一个字符，但是如果在不进行读取的时候机器也能改变状态，将会怎样呢？

在这一节，我们将探索这些想法，在对有限自动机的能力稍做调整之后，看看是否有什么新的可能性。

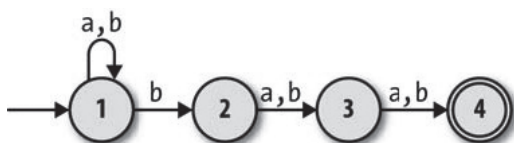
3.2.1 非确定性

假设我们想要一台有限自动机，它能接受由 a 和 b 组成的第三个字符是 b 的任意字符串。此时很容易想出一个合适的 DFA 设计：



如果想要一台机器能接受倒数第三个字符是 b 的字符串，怎么办呢？那将如何工作呢？似乎更加困难：上面的 DFA 能保证在读第三个字符的时候处于状态 3，但是一台机器无法预先知道什么时候能读到倒数第三个字符，因为在结束读取之前它不知道这个字符串有多长。甚至这样的一台 DFA 是否可能存在都不一定能立刻清楚。

但是，如果我们放松确定性的限制，并且允许规则手册对于一个状态和输入包含多条规则（或者根本没有规则），那么就可以设计一台能完成任务的机器：



这是一台*非确定性有限自动机*（NFA），对每一个输入序列不再只有一条执行路径。处于状态 1 并且读入 b 的时候，它可能会按照一条规则仍保持在状态 1，但也可能会按照另一条规则进入状态 2。反过来，一旦进入状态 4，它找不到任何规则可以遵守，因此没法再继续读取输入。一台 DFA 的下一状态总是完全由它的当前状态和输入决定，但是一台 NFA 在向下一个状态转移时会有多种可能性，而且有时候根本无法转移。

如果一台 DFA 读取一个字符串然后完全按照规则执行，并且最终终止于一个接受状态，那它就能接受这个字符串。那么对于一台 NFA 来说，什么才能表示一台 NFA 接受或者拒绝一个字符串呢？很自然的回答是，如果存在某条路径能让 NFA 按照它的*某些*规则执行并终止于一个接受状态，那它就能接受这个字符串；这就是说，即使不是*必然的*，只要终止于一个接受状态是*可能*的就可以。

例如，这台 NFA 接受字符串 'baa'，因为从状态 1 开始，有一条路径可以让这台机器读取一个 b 转移到状态 2，再读取一个 a 转移到状态 3，最后读一个 a 终止于状态 4，这是一个接受态。它还接受字符串 'bbbbb'，因为 NFA 可以在读取前两个 b 的时候，按照另一条规则执行并停留在状态 1，然后在读第三个 b 的时候使用规则转移到状态 2，再读取字符串的其他部分，并向以前那样终止于状态 4。

另一方面，没有读取 'abb' 并终止于状态 4 的方法（取决于遵照的不同规则，它最终只能终止于状态 1、2 或者 3），因此这台 NFA 不接受 'abb'。'bbabb' 也不行，它最多只能到达状态 3：如果读入第一个 b 的时候直接转移到状态 2，它将很快终止于状态 4，这样留下两个字符没有处理但是已经没有规则可用了。



能被一台特定机器接受的字符串集合称为一种*语言*：我们说这台机器*识别*了这种语言。不是所有的语言都有一台 DFA 或者 NFA 能识别它们（详见第 4 章），但那些能被有限自动机识别的语言称为*正则语言*（regular language）。

放松确定性约束已经造就了一台虚拟机器，这台虚拟机器与我们现实中熟悉的确定性机器差别很大。一台 NFA 按照可能性而不是确定性工作：我们根据*可能*发生的而不是*将要*发生的来讨论它的行为。这似乎很强大，但是这样的机器在现实世界中如何工作呢？初看上去，现实中一台 NFA 的实现需要某种预见性，要在读取输入的时候从几种可能性中做出选择：为了保留接受一个字符串的可能，示例 NFA 一定要在读到倒数第三个字符之前保持在状态 1，但它没法知道还将收到多少个字符。我们怎么用乏味又确定的 Ruby 模拟这样一台激动人心的机器呢？

在确定性计算机上模拟一台 NFA，关键是找到一种方法探索出这台机器*所有可能的执行*。这种暴力方法把所有的可能全都摆出来，以此避免了只模拟一种可能执行时所需要的“幽灵般”的预见性。一台 NFA 读到一个字符的时候，它下一步转移到什么状态只会有有限数目的可能性，因此我们模拟非确定性时可以尝试遍历所有可能，然后看它们中哪个最终到达一个接受状态。

尝试遍历所有可能时可以采用递归的方式：每当所模拟的 NFA 读取一个字符并且有多个可用的规则时，遵照其中的一条规则，然后尝试读取输入的后续部分；如果这没有让机器到达一个可接受状态，就回退到早期状态，把输入也倒回早期的位置，然后按照另一个不同的规则再次尝试；如此重复，直到某次选择的规则让机器到达一个接受状态，或者所有可能的选择进行遍历的结果都不成功为止。

还有一个策略是采用并行的方式模拟所有可能：每当机器有超过一条规则可以遵守时就创建新线程，并把需要模拟的 NFA 复制过去以便复制的每一份都能尝试一条新规则，然后观察它的结果。所有这些线程都能同时执行，每个都从它自己的输入字符串副本中读取。如果任何一个线程让机器读取了整个字符串，并且停止于一个接受状态，那么可以说这个字符串已经被接受了。

这两个实现都是可行的，但是有些复杂和低效。我们模拟的 DFA 非常简单，而且能读取单个字符并报告这台机器是否处于一个接受状态，因此要是能模拟一台有同样简单和透明的 NFA 就好了。

幸运的是，存在一个简单的方式模拟 NFA，而无需回退进程、创建线程或者预先知道所有的输入字符。事实上，就像通过跟踪一台 DFA 的当前状态来模拟它一样，我们可以通过跟踪一台 NFA 当前所有*可能*的状态模拟一台简单的 NFA。这样比模拟要转移到不同方向的多份 NFA 更简单更高效，且最终能完成同样的事情。之前，如果我们模拟很多份独立的机器，那么只需要注意它们每一个都处于什么状态，但处于同样状态的机器是完全无法分辨的[2]，因此我们把所有可能都压缩到一台机器上并询问“到现在为止它*可能*处于什么状态”，这样就不会失去任何东西了。

举个例子，让我们演练一下在读取字符串 `'bab'` 时示例 NFA 会发生什么。

- 在 NFA 读取任何输入之前，它肯定处于起始状态，也就是状态 1。
- 读取第一个字符 `b`。在状态 1，有一个 `b` 的规则可以让 NFA 停留在状态 1，并且还有一个 `b` 的规则可以把它转移到状态 2，这样我们知道之后它可能处于状态 1 或者状态 2。这些都不是接受状态，这表明 NFA 不可能通过读字符串 `'b'` 到达一个接受状态。
- 读取第二个字符 `a`。如果它处于状态 1，那么只有一个 `a` 的规则可以用，这让它继续处于状态 1；如果它处于状态 2，就只能按照 `a` 的规则转移到状态 3。它一定会终止于状态1或者状态3,而这些又都不是接受状态,因此没有方法让字符串 `'ba'` 被这台机器接受。
- 读取第三个字符 `b`。如果它处于状态 1，那么就像以前一样，继续处于状态 1 或者转移到状态 2；如果它处于状态 3，那就一定会转移到状态 4。
- 现在我们知道 NFA 在读取整个输入字符串之后可能处于状态 1、状态 2 或者状态 4。状态 4 是一个接受状态，并且我们的模拟表明一定有*某种*方式让机器通过读取那个字符串到达状态 4，因此这个 NFA *确实*能接受 `'bab'`。

这个模拟策略很容易转换成代码。首先，我们需要一个适合存储 NFA 规则的规则手册。当我们询问 DFA 规则手册处于特定状态的 DFA 读到一个特定的字符之后下一步应该转移到何处时，它总会返回一个状态。但是，NFA 规则手册需要回答一个不同的问题：在 NFA 处于几种可能状态之一时，它读取到一个特定的字符，可能的下一个状态是什么呢？实现如下：

```
require 'set'

class NFARulebook < Struct.new(:rules)
  def next_states(states, character)
    states.flat_map { |state| follow_rules_for(state, character) }.to_set
  end

  def follow_rules_for(state, character)
    rules_for(state, character).map(&:follow)
  end
```

注 2：一台有限自动机不记录自己的历史，除了它的当前状态也不做任何存储，因此处于同样状态的两台相同的机器不管出于什么目的都是可以互换的。

```
  def rules_for(state, character)
    rules.select { |rule| rule.applies_to?(state, character) }
  end
end
```



为了存储由 #next_states 返回的可能状态，我们使用 Ruby 标准库中的 Set 类。我们本来可以使用 Array 类，但是 Set 类有三个有用的特性。

(1) 它自动去除重复元素。Set[1,2,2,3,3,3] 与 Set[1,2,3] 等价。

(2) 它不关心元素的顺序。Set[3,2,1] 与 Set[1,2,3] 等价。

(3) 它提供标准的集合操作，比如交集（#&）、并集（#+）以及子集测试（#subset?）。

第一个特性很有用，因为“这台 NFA 处于状态 3 或者状态 3”这句话是讲不通的，而且返回一个 Set 能确保永远不会包含任何重复数据。其他两个特性的益处将在稍后显现。

我们可以创建一个非确定性的规则手册并向它提问：

```
>> rulebook = NFARulebook.new([
     FARule.new(1, 'a', 1), FARule.new(1, 'b', 1), FARule.new(1, 'b', 2),
     FARule.new(2, 'a', 3), FARule.new(2, 'b', 3),
     FARule.new(3, 'a', 4), FARule.new(3, 'b', 4)
   ])
=> #<struct NFARulebook rules=[...]>
>> rulebook.next_states(Set[1], 'b')
=> #<Set: {1, 2}>
>> rulebook.next_states(Set[1, 2], 'a')
=> #<Set: {1, 3}>
>> rulebook.next_states(Set[1, 3], 'b')
=> #<Set: {1, 2, 4}>
```

下一步就是实现一个 NFA 类来表示这台模拟的机器：

```
class NFA < Struct.new(:current_states, :accept_states, :rulebook)
  def accepting?
    (current_states & accept_states).any?
  end
end
```



方法 NFA#accepting? 通过检查是否在 current_states 和 accept_states 的交集里存在任何状态来完成自己的工作——也就是说，检查当前的可能状态是否也是一个接受状态。

这个 NFA 类与我们之前的 DFA 类非常相似。不同的是，它有一个当前可能的状态集合 current_states 而不是只有一个当前的确定状态 current_state，因此如果 current_states

里有一个是接受状态，就说它处于接受状态：

```
>> NFA.new(Set[1], [4], rulebook).accepting?
=> false
>> NFA.new(Set[1, 2, 4], [4], rulebook).accepting?
=> true
```

就像 DFA 类一样，我们可以实现一个 #read_character 方法读取输入中的一个字符，以及一个 #read_string 方法可以按顺序读取几个字符：

```
class NFA
  def read_character(character)
    self.current_states = rulebook.next_states(current_states, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

这些方法实际上与它们对应的 DFA 几乎完全相同，只是在 #read_character 中使用了 current_states 和 next_states，而不是 current_state 和 next_state。

困难的工作结束了。现在我们可以启动一个模拟的 NFA，给它传入字符，并且询问它目前的输入是否已经被接受：

```
>> nfa = NFA.new(Set[1], [4], rulebook); nfa.accepting?
=> false
>> nfa.read_character('b'); nfa.accepting?
=> false
>> nfa.read_character('a'); nfa.accepting?
=> false
>> nfa.read_character('b'); nfa.accepting?
=> true
>> nfa = NFA.new(Set[1], [4], rulebook)
=> #<struct NFA current_states=#<Set: {1}>, accept_states=[4], rulebook=...>
>> nfa.accepting?
=> false
>> nfa.read_string('bbbbb'); nfa.accepting?
=> true
```

就像我们在使用 DFA 类时看到的那样，可以很方便地使用一个 NFADesign 对象根据需要自动生产新的 NFA 实例，而不是手工创建它们：

```
class NFADesign < Struct.new(:start_state, :accept_states, :rulebook)
  def accepts?(string)
    to_nfa.tap { |nfa| nfa.read_string(string) }.accepting?
  end
```

```
  def to_nfa
    NFA.new(Set[start_state], accept_states, rulebook)
  end
end
```

这让同一台 NFA 检查不同的字符串更容易：

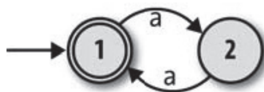
```
>> nfa_design = NFADesign.new(1, [4], rulebook)
=> #<struct NFADesign start_state=1, accept_states=[4], rulebook=...>
>> nfa_design.accepts?('bab')
=> true
>> nfa_design.accepts?('bbbbb')
=> true
>> nfa_design.accepts?('bbabb')
=> false
```

就是这样了。我们已经通过模拟一台非同寻常的非确定性机器的所有可能执行，并构建了它的一个简单实现。非确定性是一个设计更复杂有限自动机的非常方便的工具，因此我们很幸运能把 NFA 投入实际使用而不只是把它作为理论中的珍品。

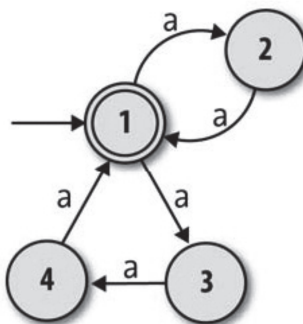
3.2.2 自由移动（free move）

我们已经看到，对确定性约束的放松带来了设计机器的新方式，我们不再需要殚精竭力地去实现它们了。为了得到更多的设计自由，我们还可以安全地放松哪些约束呢？

很容易设计一台 DFA，能接受长度是 2 的倍数的、由字符 a 组成的字符串（'aa'、'aaaa'……）：

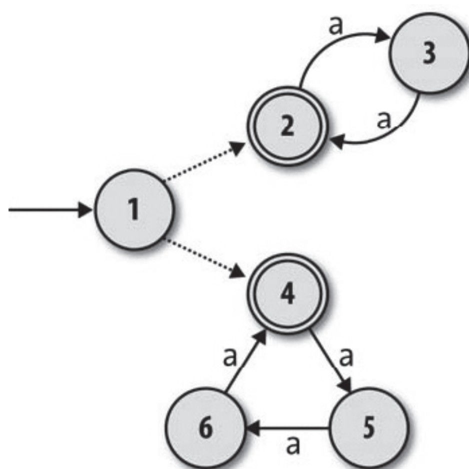


但是如何设计一台机器，让它能接受长度是 2 或 3 的倍数的字符串呢？我们知道非确定性让一台机器可以走多于一条的执行路径，因此或许可以设计一台 NFA，它有一条“2 的倍数”的路径和一条“3 的倍数”的路径。一个初步的尝试可能看起来像这个样子：



这台 NFA 的思想是，在状态 1 和状态 2 之间移动以接受像 'aa' 和 'aaaa' 这样的字符串，在状态 1、状态 3 和状态 4 之间移动以接受像 'aaa' 和 'aaaaaaaaa' 这样的字符串。这工作得很好，但问题是这台机器还会接受字符串 'aaaaa'，因为它可以从状态 1 转移到状态 2 然后读完前两个字符的时候回到状态 1，再在状态 3 和状态 4 之间转移，之后在读完接下来的三个字符之后回到状态 1，终止于一个接受状态，即使这个字符串的长度不是 2 或者 3 的倍数。[3]

这次，一台 NFA 是否能完成这个工作还不是很明显，但是我们可以引入一个叫作*自由移动*的机器特性来解决此问题。这些规则让机器无需读取任何输入就能自发遵照执行，并且它们在这儿提供帮助是因为能让 NFA 在两组状态之间做一个初步选择：



自由移动表示成从状态 1 到状态 2 和状态 4 的无标记虚线箭头。机器仍然接受字符串 'aaaa'，它会先自发地转移到状态 2，然后随着读取输入在状态 2 和状态 3 之间转移。类似地，如果它开始先自由移动到状态 4 也能接受 'aaaaaaaaa'。但是现在它没法接受字符串 'aaaaa' 了：不管做任何可能的执行，它都一定要从到状态 2 或者状态 4 的转移开始，而且一旦选择了其中一条路径转移之后，就没法退回来了。一旦处于状态 2，就只能接受一个长度是 2 的倍数的字符串，同样一旦处于状态 4，就只能接受长度是 3 的倍数的字符串。

如何用 Ruby 模拟 NFA 中的自由移动呢？当然，是保持在状态 1、自发地转移到状态 2，还是自发地转移到状态 4，这些新选择并不比已有的非确定性奇怪多少，并且我们的实现能够用类似的方式处理它。我们已经有了一台模拟机一次可以有多个可能状态的思想，因此只需要拓展那些可能的状态，把通过执行一次或者多次自由移动能到达的状态包括进

注 3：实际上，这台 NFA 接受字符 a 组成的任何字符串，但只有一个字符的字符串 'a' 除外。

来。在这种情况下，“机器从状态 1 开始”的真正意思是：在没有读取任何输入之前，它可能处于状态 1、2 或 4。

首先，我们需要一种用 Ruby 表示自由移动的方法。最简单的方法就是使用正常的 FARule 实例，只是在一个字符的位置上填上一个 nil。NFARulebook 的现有实现将像处理其他任何字符一样处理 nil，因此我们可以询问：“从状态 1，通过执行一次自由移动（而不是问：“……通过读入一个字符 a？”），能到达什么状态？”

```
>> rulebook = NFARulebook.new([
     FARule.new(1, nil, 2), FARule.new(1, nil, 4),
     FARule.new(2, 'a', 3),
     FARule.new(3, 'a', 2),
     FARule.new(4, 'a', 5),
     FARule.new(5, 'a', 6),
     FARule.new(6, 'a', 4)
   ])
=> #<struct NFARulebook rules=[...]>
>> rulebook.next_states(Set[1], nil)
=> #<Set: {2, 4}>
```

下一步需要一些辅助代码帮助找到从一个特定集合的状态开始，通过自由移动所能到达的所有状态。这些代码只能反复自由移动，因为只要存在从当前状态出发的自由移动，一台 NFA 就可以多次自发改变状态。可以把它很方便地放到 NFARulebook 类的一个方法里：

```
class NFARulebook
  def follow_free_moves(states)
    more_states = next_states(states, nil)

    if more_states.subset?(states)
      states
    else
      follow_free_moves(states + more_states)
    end
  end
end
```

NFARulebook#follow_free_moves 以递归的方式查找越来越多的状态，这些状态能从一个给定的集合通过自由移动到达。再也找不到时，即由 next_states(states,nil) 找到的每一个状态都已经包含在 states 里时，它就返回找到的所有状态。[4]

以下代码正确地识别出 NFA 在读取任何输入之前的可能状态：

```
>> rulebook.follow_free_moves(Set[1])
=> #<Set: {1, 2, 4}>
```

现在通过覆盖 NFA#current_states 已有的实现（就像覆盖 Struct 提供的方法一样），我们

注 4：确切地说，这个过程计算了“通过自由移动增加更多状态”函数的定点。

把对自由移动的支持加入到 NFA 当中。新的实现将与 `NFARulebook#follow_free_moves` 挂钩，并确保自动机当前可能的状态总是包含通过自由移动能到达的任何状态：

```
class NFA
  def current_states
    rulebook.follow_free_moves(super)
  end
end
```

因为其他所有 `NFA` 方法都是通过调用 `#current_states` 访问当前可能状态的集合，所以这种透明性让我们不必改动 `NFA` 代码的其他部分就能支持自由移动。

这就全部完成了。现在模拟支持自由移动了，而且现在能看看哪些字符串能被我们的 NFA 接受了：

```
>> nfa_design = NFADesign.new(1, [2, 4], rulebook)
=> #<struct NFADesign ...>
>> nfa_design.accepts?('aa')
=> true
>> nfa_design.accepts?('aaa')
=> true
>> nfa_design.accepts?('aaaaa')
=> false
>> nfa_design.accepts?('aaaaaa')
=> true
```

自由移动实现起来非常简单，并且在非确定性的基础之上给了我们额外的设计自由。



本章中有一些非传统术语。有限自动机读取的字符通常叫作*符号*（symbol），状态之间移动的规则叫作*转移*（transition），组成一台机器的规则集合叫作*转移函数*（有时候也叫 NFA 的*转移关系*）而不是规则手册。因为表示空字符串的数学符号是希腊字母 ε，能自由移动的 NFA 称为 NFA-ε，自由移动本身通常称为 *ε 转移*。

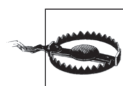
3.3 正则表达式

我们已经看到非确定性和自由移动增强了有限自动机的表达能力，而且不会干扰我们对有限自动机的模拟。在这一节，我们将会看到这些特性一个重要的实际应用：正则表达式匹配。

*正则表达式*提供了书写*模式*的语言，字符串可以按照这个模式进行匹配。下面是一些正则表达式的例子。

- `hello`，只能匹配字符串 `'hello'`。
- `hello|goodbye`，能匹配字符串 `'hello'` 和 `'goodbye'`。

- (hello)*，匹配字符串 'hello'、'hellohello'、'hellohellohello' 等，也与空字符串匹配。



在这一章里，我们把正则表达式看成是与整个字符串进行匹配。真实世界中的正则表达式实现通常与部分字符串匹配，如果要求与整个字符串匹配的话，则应该使用额外的语法。

例如，我们的正则表达式 hello|goodbye 在 Ruby 中应该写成 /\A(hello|goodbye)\z/，这确保任何匹配都固定在字符串的开始（\A）和结尾（\z）之间。

给定一个正则表达式和一个字符串，我们如何写程序决定这个字符串是否与那个表达式匹配呢？大多数的编程语言，包括 Ruby 在内，已经内建了对正则表达式的支持，但是这样的支持是如何工作的呢？如果语言没有支持正则表达式，我们如何使用 Ruby 实现它们呢？

有限自动机完全适合这个工作。就像我们即将看到的，把任何正则表达式转成一个等价的 NFA 是可能的——每一个与正则表达式匹配的字符串都能被这台 NFA 接受，反过来也一样——把字符串输入给一台模拟的 NFA 看它是否能被接受，从而判断字符串是否与正则表达式匹配。用第 2 章的话说，我们可以把这个看成是为正则表达式提供了一种指称语义：我们不一定知道如何直接执行一个正则表达式，但是可以展示如何把它表示成一台 NFA，并且因为有了 NFA 的操作语义（“通过读取字符然后执行规则改变状态”），所以可以执行这个指称（denotation）实现同样的结果。

3.3.1　语法

让我们明确一下“正则表达式”是什么意思。下面是两种极其简单的正则表达式，它们已经没法更简单了。

- 一个空的正则表达式。与空字符匹配，没有别的可匹配的了。
- 一个只含有一个字符的正则表达式。例如，a 和 b 是分别只能匹配 'a' 和 'b' 的正则表达式。

有了这几种简单的模式之后，我们有三种方式可以把它们结合起来构造更复杂的表达式。

- 连接两个模式。我们可以把正则表达式 a 和 b 连接起来得到正则表达式 ab，它只与字符串 'ab' 匹配。
- 在两个模式之间选择，使用运算符 | 把它们联结起来。我们可以把正则表达式 a 或 b 联结在一起得到 a|b，它与字符串 'a' 和 'b' 匹配。
- 重复一个模式零次或者多次，写法是加上运算符 * 作为后缀。我们可以给正则表达式 a 加上后缀得到 a*，它与字符串 'a'、'aa'、'aaa' 等匹配，当然也与空字符串 '' 匹配（也就是说重复零次）。



现实中的正则表达式引擎（比如构建到 Ruby 当中的），支持更多的特性。为了简单起见，我们不会尝试实现这些额外的特性，它们中有很多从学术上讲多余，只是为了方便才提供的。

例如，省略运算符 ? 和 + 没有什么太大区别，因为它们的作用（分别为“重复一或者零次”和“重复一或者多次”）很容易使用已有的特性实现：正则表达式 ab? 可以重写成 ab|a，而模式 ab+ 与 abb* 匹配同样的字符串。其他计数重复（如 a{2,5}）和字符组（如 [abc]）等方便的特性也是这样。

捕获组（capture group）、反向引用（backreference）以及先行 / 后行断言（lookahead/lookbehind assertion）这样的高级特性已经超出了本章的讲述范围。

为了使用 Ruby 实现这个语法，我们可以为每类正则表达式定义一个类，并使用这些类的实例表示任何正则表达式的抽象语法树，就像在第 2 章里处理 Simple 表达式一样：

```
module Pattern
  def bracket(outer_precedence)
    if precedence < outer_precedence
      '(' + to_s + ')'
    else
      to_s
    end
  end

  def inspect
    "/#{self}/"
  end
end

class Empty
  include Pattern

  def to_s
    ''
  end

  def precedence
    3
  end
end

class Literal < Struct.new(:character)
  include Pattern

  def to_s
    character
  end

  def precedence
    3
```

```
  end
end

class Concatenate < Struct.new(:first, :second)
  include Pattern

  def to_s
    [first, second].map { |pattern| pattern.bracket(precedence) }.join
  end

  def precedence
    1
  end
end

class Choose < Struct.new(:first, :second)
  include Pattern

  def to_s
    [first, second].map { |pattern| pattern.bracket(precedence) }.join('|')
  end

  def precedence
    0
  end
end

class Repeat < Struct.new(:pattern)
  include Pattern

  def to_s
    pattern.bracket(precedence) + '*'
  end

  def precedence
    2
  end
end
```



在算术表达式中乘法对它参数的绑定比加法要更紧（1+2×3 等于 7，而不是 9），同样，这个约定也适用于正则表达式的语法，它的 * 运算符也比串联运算符绑定得更紧，而串联运算符又比 | 运算符绑定得紧。例如，在正则表达式 abc* 中，* 只会应用到 c 上（'abc'、'abcc'、'abccc'……），而为了让它能应用到整个 abc 上（'abc'、'abcabc'），需要加上括号写成 (abc)*。

语法类的实现 #to_s 和 Pattern#bracket 方法一起，会在必要的时候自动插入括号，这样在查看一棵抽象语法树的简单字符串表示时，我们也能知道它的结构信息。

有了这些类，我们就可以手工构建表示正则表达式的树：

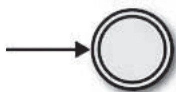
```
>> pattern =
     Repeat.new(
       Choose.new(
         Concatenate.new(Literal.new('a'), Literal.new('b')),
         Literal.new('a')
       )
     )
=> /(ab|a)*/
```

当然，在实际的实现中，我们不会手工构建这些树，而会使用语法解析器构建它们；可以参考 3.3.3 节。

3.3.2　语义

既然我们可以把正则表达式语法表示成 Ruby 对象组成的树，那么如何把这个语法转换成 NFA 呢？

我们需要知道每个语法类的实例应该如何转换成 NFA。转换起来最简单的类是 `Empty`，应该总是把它转换成一个状态的 NFA，这个 NFA 只接受空字符串：



类似地，我们应该把任何单字符的模式转换成只接受包含那个字符的、单字符串的 NFA。下面是模式 `a` 的 NFA：



为 `Empty` 和 `Literal` 实现 `#to_nfa_design` 方法来生成这些 NFA 相当容易：

```
class Empty
  def to_nfa_design
    start_state = Object.new
    accept_states = [start_state]
    rulebook = NFARulebook.new([])

    NFADesign.new(start_state, accept_states, rulebook)
  end
end

class Literal
  def to_nfa_design
    start_state = Object.new
    accept_state = Object.new
    rule = FARule.new(start_state, character, accept_state)
    rulebook = NFARulebook.new([rule])
```

```
    NFADesign.new(start_state, [accept_state], rulebook)
  end
end
```



3.1.4 节提到过，用 Ruby 对象实现自动机时，状态对象彼此之间一定要能区分。这里没有使用数字（如 Fixnum 实例）作为状态，而是使用了新创建的 Object 实例。

这是为了每一个 NFA 都能有它自己独一无二的状态，以便把小的机器组合成大的机器，而不会意外把它们的状态也进行归并。例如，如果两个不同的 NFA 都使用 Ruby 的 Fixnum 对象 1 作为状态，在保持它们两个状态独立的情况下，它们不能合到一起。但是我们将来会需要能进行这样的合并，以便能实现更复杂的正则表达式。

类似地，我们不会继续在图上为状态打标记，这样以后把图连到一起时也不用重新对其进行标记。

可以检查由 Empty 和 Literal 正则表达式生成的 NFA 能否接受我们想要它接受的字符串：

```
>> nfa_design = Empty.new.to_nfa_design
=> #<struct NFADesign ...>
>> nfa_design.accepts?('')
=> true
>> nfa_design.accepts?('a')
=> false
>> nfa_design = Literal.new('a').to_nfa_design
=> #<struct NFADesign ...>
>> nfa_design.accepts?('')
=> false
>> nfa_design.accepts?('a')
=> true
>> nfa_design.accepts?('b')
=> false
```

这里有机会可以把 #to_nfa_design 封装进 #matches? 方法，让模式有一个更友好的接口：

```
module Pattern
  def matches?(string)
    to_nfa_design.accepts?(string)
  end
end
```

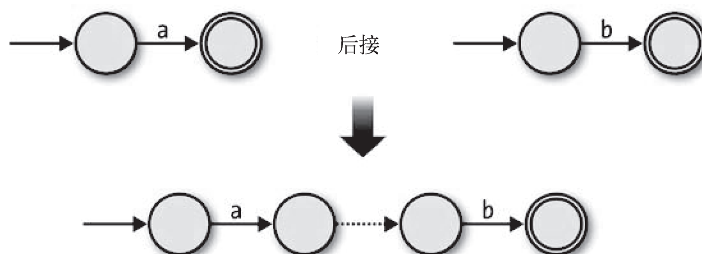
这样我们就可以直接用模式匹配字符串：

```
>> Empty.new.matches?('a')
=> false
>> Literal.new('a').matches?('a')
=> true
```

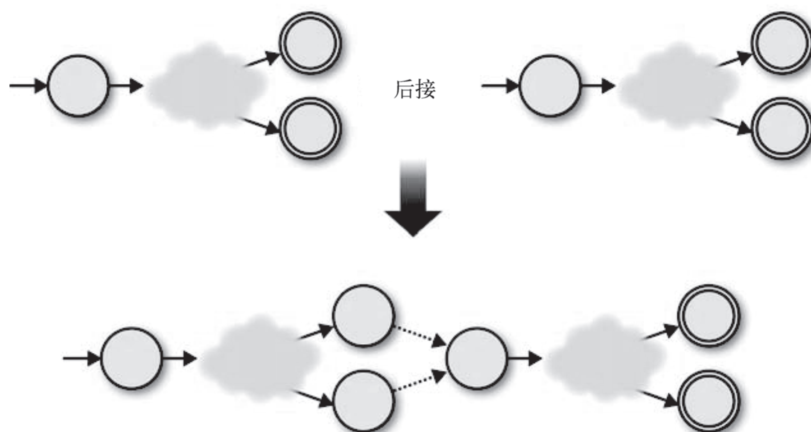
既然我们知道如何把简单的 `Empty` 和 `Literal` 正则表达式转成 NFA 了，那对 `Concatenate`（串联）、`Choose`（选择）和 `Repeat`（重复）也需要类似的进行转换。

从 `Concatenate` 开始：如果有两个已经知道如何转换成 NFA 的正则表达式，那么如何构造一个 NFA 表示这些正则表达式的串联呢？举个例子，假如能把单个字符的正则表达式 `a` 和 `b` 转换成 NFA，那怎么把 `ab` 转成一个 NFA 呢？

对于 `ab`，我们可以把两个 NFA 按顺序连接到一起，用自由移动把它们联结在一起，并且保留第二个 NFA 的接受状态：



这个技术在其他情况下也行得通。任意两个 NFA 的连接，都可以先把第一个 NFA 的每一个接受状态转成非接受状态，再通过自由移动把它与第二个 NFA 的开始状态连接。如果一串输入能让原来第一台 NFA 进入接受状态，串联起来的机器读入这串输入之后就能自发的进入到原来第二个 NFA 的起始状态，然后通过读取一串原来第二个 NFA 能接受的输入，它将到达自己的接受状态。



因此，组合机器的原材料是：

- 第一个 NFA 的起始状态；
- 第二个 NFA 的接受状态；

- 两台 NFA 的所有规则；
- 一些额外的自由移动，可以把第一台 NFA 旧的接受状态与第二个 NFA 旧的起始状态连接起来。

可以把这个想法转换成 Concatenate#to_nfa_design 的实现：

```
class Concatenate
  def to_nfa_design
    first_nfa_design = first.to_nfa_design
    second_nfa_design = second.to_nfa_design

    start_state = first_nfa_design.start_state
    accept_states = second_nfa_design.accept_states
    rules = first_nfa_design.rulebook.rules + second_nfa_design.rulebook.rules
    extra_rules = first_nfa_design.accept_states.map { |state|
      FARule.new(state, nil, second_nfa_design.start_state)
    }
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

这段代码首先把第一和第二个正则表达式转换成 NFADesign，然后把它们的状态和规则用合适的方式组合到一起构成新的 NFADesign。ab 这种简单的情况是没有问题的：

```
>> pattern = Concatenate.new(Literal.new('a'), Literal.new('b'))
=> /ab/
>> pattern.matches?('a')
=> false
>> pattern.matches?('ab')
=> true
>> pattern.matches?('abc')
=> false
```

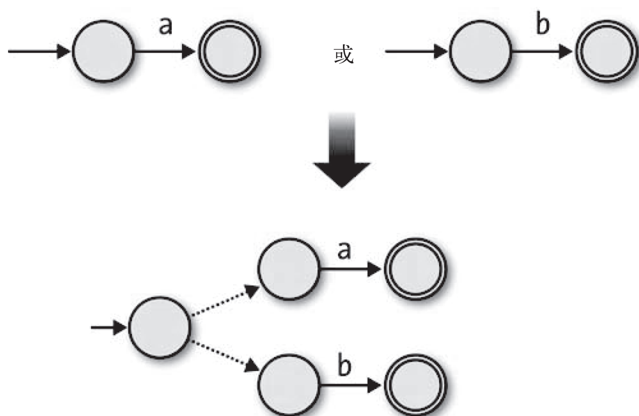
这个转换过程是递归的（Concatenate#to_nfa_design 对其他对象调用 #to_nfa_design），因此对于像 abc 这样的更深嵌套的正则表达式也能正常工作，这种情况下将包含两次串联（a 与 b 串联然后与 c 串联）：

```
>> pattern =
     Concatenate.new(
       Literal.new('a'),
       Concatenate.new(Literal.new('b'), Literal.new('c'))
     )
=> /abc/
>> pattern.matches?('a')
=> false
>> pattern.matches?('ab')
=> false
>> pattern.matches?('abc')
=> true
```

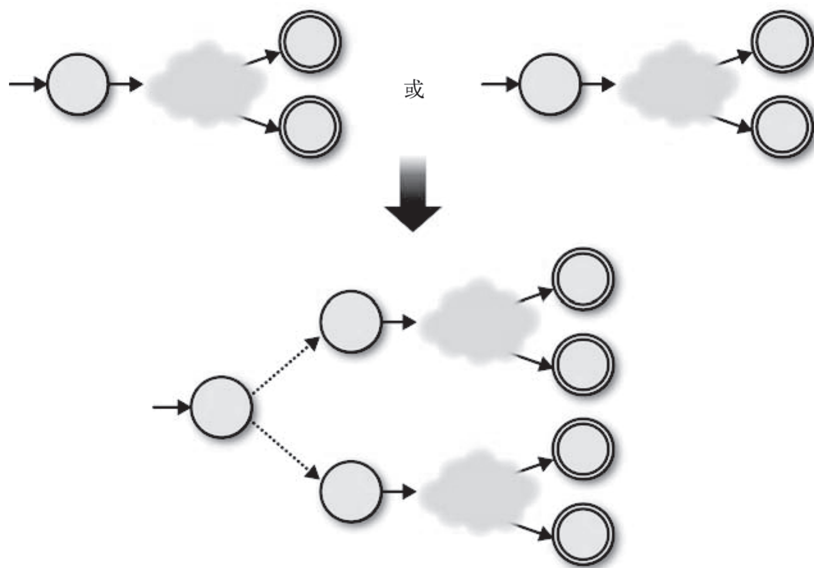


这又是一个组合型指称语义的例子：复合正则表达式的 NFA 指称由它每一部分 NFA 的指称组成。

我们可以使用同样的策略把 `Choose` 表达式转成一台 NFA。在最简单的情况下，正则表达式 `a` 和 `b` 的 NFA 能结合起来构造成正则表达式 `a|b` 的 NFA，方法是增加一个新的起始状态并使用自由移动把它与两台原始机器之前的起始状态连接起来：



在 `a|b` NFA 读取任何输入之前，它可以自由移动进入任何一个原始机器的起始状态，再从这个状态开始读取 `'a'` 或者 `'b'` 从而到达一个接受状态。通过增加一个新的起始状态和两个自由移动，把任意两台机器连到一起很简单：



在这种情况下，组合机器的原材料是：

- 一个新的起始状态；
- 两台 NFA 的所有接受状态；
- 两台 NFA 的所有规则；
- 两个额外的自由移动，可以把新的起始状态与 NFA 旧的起始状态连接起来。

实现 Choose#to_nfa_design 仍然不难：

```
class Choose
  def to_nfa_design
    first_nfa_design = first.to_nfa_design
    second_nfa_design = second.to_nfa_design

    start_state = Object.new
    accept_states = first_nfa_design.accept_states + second_nfa_design.accept_states
    rules = first_nfa_design.rulebook.rules + second_nfa_design.rulebook.rules
    extra_rules = [first_nfa_design, second_nfa_design].map { |nfa_design|
      FARule.new(start_state, nil, nfa_design.start_state)
    }
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

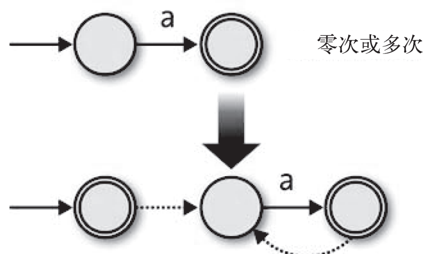
这个实现很好：

```
>> pattern = Choose.new(Literal.new('a'), Literal.new('b'))
=> /a|b/
>> pattern.matches?('a')
=> true
>> pattern.matches?('b')
=> true
>> pattern.matches?('c')
=> false
```

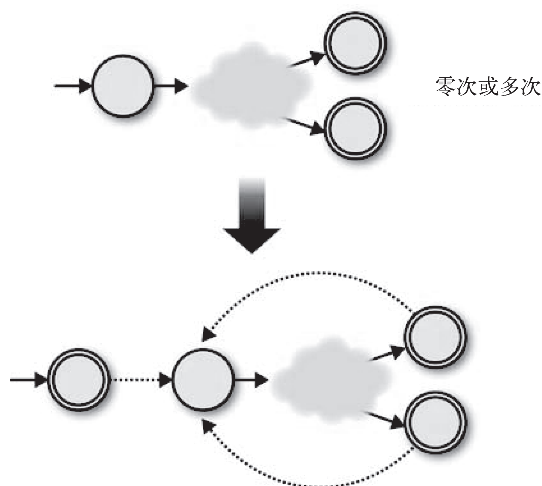
最后，我们开始讨论 Repeat：如何把与一个字符串匹配的 NFA，转换成能匹配同一个字符串重复零次或者更多次的 NFA 呢？我们为 a* 构造一个 NFA，其开头是一个 a 对应的 NFA，然后做两个补充：

- 从它的接受状态到开始状态增加一个自由移动，这样它就可以与多于一个 'a' 匹配了；
- 增加一个可自由移动到旧的开始状态的新状态，并且使其作为接受状态，这样它就可以匹配空字符串了。

图示如下：



从旧的接受状态得到旧的起始状态的自由移动，能让机器进行多次匹配而不是只匹配一次（'aa'、'aaa' 等），并且新的起始状态允许它匹配空字符串而不会影响它能接受的其他字符串[5]。对任何的 NFA 我们都可以一样处理，只要通过自由移动把每一个旧的接受状态和旧的起始状态连接起来即可：



这次我们需要：

- 一个新的起始状态，它也是一个接受状态；
- 旧的 NFA 中所有的接受状态；
- 旧的 NFA 中所有的规则；
- 一些额外的自由移动，把旧 NFA 的每一个接受状态与旧的起始状态连接起来；
- 另一些自由移动，把新的起始状态与旧的起始状态连接起来。

让我们把这些转换成代码：

注 5：在这种简单的情况下，我们可以只把原始的起始状态转成一个接受状态，而不增加新状态。但是在更复杂的情况下（例如 (a*b)*），这种技术可能会产生一台接受除了空字符串外其他一些不想要字符串的机器。

```
class Repeat
  def to_nfa_design
    pattern_nfa_design = pattern.to_nfa_design

    start_state = Object.new
    accept_states = pattern_nfa_design.accept_states + [start_state]
    rules = pattern_nfa_design.rulebook.rules
    extra_rules =
      pattern_nfa_design.accept_states.map { |accept_state|
        FARule.new(accept_state, nil, pattern_nfa_design.start_state)
      } +
      [FARule.new(start_state, nil, pattern_nfa_design.start_state)]
    rulebook = NFARulebook.new(rules + extra_rules)

    NFADesign.new(start_state, accept_states, rulebook)
  end
end
```

然后检查结果：

```
>> pattern = Repeat.new(Literal.new('a'))
=> /a*/
>> pattern.matches?('')
=> true
>> pattern.matches?('a')
=> true
>> pattern.matches?('aaaa')
=> true
>> pattern.matches?('b')
=> false
```

既然每个正则表达式语法类都已经有了 #to_nfa_design 实现，下面就可以构建复杂的模式并用它们匹配字符串了：

```
>> pattern =
     Repeat.new(
       Concatenate.new(
         Literal.new('a'),
         Choose.new(Empty.new, Literal.new('b'))
       )
     )
=> /(a(|b))*/
>> pattern.matches?('')
=> true
>> pattern.matches?('a')
=> true
>> pattern.matches?('ab')
=> true
>> pattern.matches?('aba')
=> true
>> pattern.matches?('abab')
=> true
>> pattern.matches?('abaab')
=> true
>> pattern.matches?('abba')
=> false
```

这个结果很好。我们从模式的语法开始，然后展示如何把任意模式转换成一台 NFA，而 NFA 是我们已经知道如何执行的抽象机器，这样就拥有了这种语法的语义。再配上一个语法解析器，我们就有了一种实用的方法，可以读取正则表达式并决定它是否与某个特定的字符串匹配。对这种方法自由移动非常有用，因为它们能把小一些的机器组合成更大的机器，并且不会影响其中任何组成部分的行为。



现实中多数正则表达式实现（如 Ruby 使用的 Onigmo 库）的工作方式都不是照字面把模式编译到有限自动机然后模拟它们执行。尽管这种方法在对字符串进行正则表达式匹配时快而且高效，但是在支持更高级的特性，如捕获组（capture groups）和先行 / 后行断言（lookahead/lookbehind assertions）时，会困难得多。因此，大多数的库都使用某种*回溯算法*（backtracking algorithm）更直接地处理正则表达式，而不是把它们转换成有限自动机。

Russ Cox 的 RE2 库（http://code.google.com/p/re2/）是一个产品质量级别的 C++ 正则表达式实现，它不把模式编译成自动机[6]，而 Pat Shaughnessy 已经写了一篇很详细的博客（http://patshaughnessy.net/2012/4/3/exploring-rubys-regular-expression-algorithm），来探索 Ruby 正则表达式如何工作。

3.3.3 解析

我们几乎构建了一个完整的（虽然很基本）正则表达式实现。唯一缺失的是一个模式语法的语法解析器：如果我们只需要写 `(a(|b))*` 而不是通过 `Repeat.new(Concatenate.new(Literal.new('a'), Choose.new(Empty.new, Literal.new('b'))))` 手工地构建出抽象语法树就方便多了。我们在 2.6 节中看到使用 Treetop 生成一个语法解析器并不困难，它能把原始语法自动转换成一个 AST（抽象语法树），因此下面也这样做来完成我们的实现。

下面是一个简单正则表达式的 Treetop 语法：

```
grammar Pattern
  rule choose
    first:concatenate_or_empty '|' rest:choose {
      def to_ast
        Choose.new(first.to_ast, rest.to_ast)
      end
    }
    /
    concatenate_or_empty
  end

  rule concatenate_or_empty
    concatenate / empty
  end
```

注 6：RE2 的口号是“一个高效的、条理化的正则表达式库”，这很难反驳。

```
  rule concatenate
    first:repeat rest:concatenate {
      def to_ast
        Concatenate.new(first.to_ast, rest.to_ast)
      end
    }
    /
    repeat
  end

  rule empty
    '' {
      def to_ast
        Empty.new
      end
    }
  end

  rule repeat
    brackets '*' {
      def to_ast
        Repeat.new(brackets.to_ast)
      end
    }
    /
    brackets
  end

  rule brackets
    '(' choose ')' {
      def to_ast
        choose.to_ast
      end
    }
    /
    literal
  end

  rule literal
    [a-z] {
      def to_ast
        Literal.new(text_value)
      end
    }
  end
end
```



规则的顺序又一次反映了每一个运算符的优先级：运算符的优先级从上到下越来越高，| 运算符的绑定最宽松，因此 choose 规则在最前面。

现在我们分析一个正则表达式，把它转换成一个抽象语法树，并使用它匹配字符串所需要的条件已经全部俱备：

```
>> require 'treetop'
=> true
>> Treetop.load('pattern')
=> PatternParser
>> parse_tree = PatternParser.new.parse('(a(|b))*')
=> SyntaxNode+Repeat1+Repeat0 offset=0, "(a(|b))*" (to_ast,brackets):
     SyntaxNode+Brackets1+Brackets0 offset=0, "(a(|b))" (to_ast,choose):
       SyntaxNode offset=0, "("
       SyntaxNode+Concatenate1+Concatenate0 offset=1, "a(|b)" (to_ast,first,rest):
         SyntaxNode+Literal0 offset=1, "a" (to_ast)
         SyntaxNode+Brackets1+Brackets0 offset=2, "(|b)" (to_ast,choose):
           SyntaxNode offset=2, "("
           SyntaxNode+Choose1+Choose0 offset=3, "|b" (to_ast,first,rest):
             SyntaxNode+Empty0 offset=3, "" (to_ast)
             SyntaxNode offset=3, "|"
             SyntaxNode+Literal0 offset=4, "b" (to_ast)
           SyntaxNode offset=5, ")"
       SyntaxNode offset=6, ")"
     SyntaxNode offset=7, "*"
>> pattern = parse_tree.to_ast
=> /(a(|b))*/
>> pattern.matches?('abaab')
=> true
>> pattern.matches?('abba')
=> false
```

3.4　等价性

本章已经描述了确定性状态机的思想，并且为它增加了更多特性。首先是非确定性，在设计机器时它能提供很多可能的执行路径。还有自由移动，它让非确定性的机器无需读取任何输入就可以改变状态。

非确定性和自由移动让设计有限状态机执行特定的工作更容易——我们已经看到它们在把正则表达式转换成状态机时非常有用——但它们为我们做了什么标准 DFA 不能做的事情吗？

把任何非确定性有限自动机转成接受完全相同字符串的确定性自动机是可能的。考虑到一台 DFA 的额外约束，这可能有些令人吃惊。但在思考一下我们对两种机器执行的模拟方式之后，这就能讲得通了。

假如我们要模拟一台特定 DFA 的行为。对这个假想 DFA 读取一个特定字符序列的模拟可能会是这样：

- 机器读取任何输入之前，它处于状态 1；
- 机器读取字符 `'a'`，那么它现在处于状态 2；
- 机器读取字符 `'b'`，那么它现在处于状态 3；
- 不再有输入，而且状态 3 是一个接受状态，所以字符串 `'ab'` 已经被接受。

这里有一些很微妙的东西：模拟在重新创造着 DFA 的行为。在我们的例子里，模拟是运行在一台真实计算机上的 Ruby 程序，而 DFA 则是无法运行的一台抽象机器，因为它根本不存在。每当假想的 DFA 改变状态的时候，正在运行的模拟也要改变——因此才称其为模拟。

很难把 DFA 和它的模拟分开，因为它们都是确定性的，而且它们的状态完全匹配：DFA 处于状态 2 的时候，模拟也处于一个能表明"这台 DFA 处于状态 2"的状态。在我们的 Ruby 模拟中，这个模拟状态实际上就是 `DFA` 实例的 `current_state` 属性值。

尽管在处理非确定性和自动移动时有额外的开销，但对一个假想的 NFA 读取字符串进行模拟并没有什么大的不同。

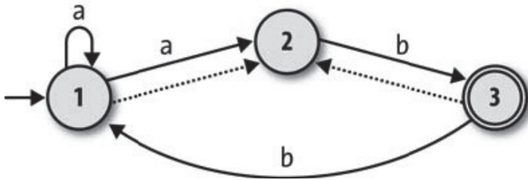
- 机器读取任何输入之前，它可能处于状态 1 或者状态 3。[7]
- 机器读取字符 `c`，那么现在它可能处于状态 1、3 或者 4 中的一个。
- 机器读取字符 `d`，那么现在它可能处于状态 2 或者状态 5 中的一个。
- 不再有输入，并且状态 5 是一个接受状态，因此字符串 `'cd'` 已经被接受。

模拟的状态与 NFA 的状态不一样，这一点此时更容易看出来。事实上，在模拟的每一点上，我们一直都无法确定 NFA 那时处于什么状态。但是模拟本身仍然是确定性的，因为它的状态能够适应这种不确定性。在 NFA 可能处于状态 1、3 或者 4 中一个的时候，我们可以肯定模拟现在处于一个表示"NFA 处于状态 1、3 或者 4"的某一个确定状态。

这两个例子的唯一真正区别是，DFA 的模拟是从一个当前状态移动到另一个，而 NFA 的模拟是从一个当前可能状态的集合移动到另一个可能状态的集合。尽管一个 NFA 的规则手册可以是非确定性的，但是对于一个给定的输入从当前状态出发移动到哪些状态，这个决定总是完全确定性的。

这种确定性意味着我们总可以构造一台 DFA 来模拟一台特定的 NFA。这台 DFA 有一个状态表示这台 NFA 的每一个可能状态的集合，并且 DFA 状态之间移动的规则对应着 NFA 的确定性模拟在它可能状态的集合之间的移动方式。这台 DFA 将能够完全模拟 NFA 的行为，并且只要为 DFA 选择合适的接受状态——根据我们的 Ruby 实现，这些将是与处于接受状态的 NFA 对应的任何状态——它也将接受同样的字符串。

让我们尝试着为一台特定的 NFA 做这种转换。以下面这个为例：

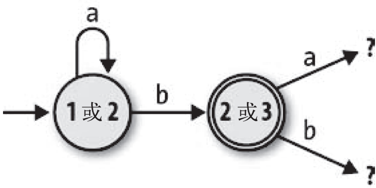


注 7：尽管一台 NFA 只有一个起始状态，但自由移动使得读取任何输入之前进入其他状态成为可能。

在没有读取任何输入之前，这台 NFA 可能处于状态 1 或者状态 2（状态 1 是起始状态，而状态 2 可以通过自由移动到达），因此模拟将从可以叫作“1 或者 2”的状态开始。从这个起点出发，根据它读到的是 a 或 b，模拟将会在不同的状态终止。

- 如果读到 a，模拟仍将保持在状态“1 或者 2”：NFA 处于状态 1 时它可以读入 a，然后或是维持在状态 1 或是进入状态 2，而从状态 2 开始，它没法再读入 a 了。
- 如果读到 b，NFA 可能会终止于状态 2 或者状态 3（从状态 1 开始），它不能再读到 b 了，但是从状态 2 开始，它可以移动到状态 3 并且还可能自由移动回状态 2；因此，我们说输入为 b 的时候，模拟将移动到叫作“2 或者 3”的状态。

通过思考一个 NFA 模拟的行为，我们可以为这个模拟构造一台状态机：





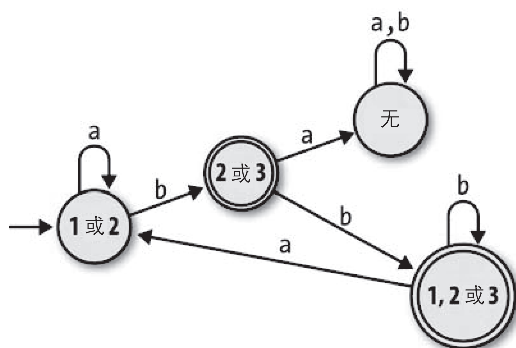
“2 或者 3”是模拟的一个接受状态，因为状态 3 是 NFA 的一个接受状态。

可以继续这个过程发现模拟的更多新状态，直到不再有新发现为止。因为原始 NFA 的状态只有有限数目的可能组合，所以最后肯定能停止发现。[8] 通过重复对示例 NFA 的发现过程，我们发现从“1 或者 2”出发然后读取 a 和 b 的序列，它的模拟只能碰到四种不同的状态组合：

如果NFA处于状态……	并且读入字符……	它可能终止于状态……
1 或 2	a	1 或 2
	b	2 或 3
2 或 3	a	无
	b	1、2 或 3
无	a	无
	b	无
1、2 或 3	a	1 或 2
	b	1、2 或 3

此表完整地描述了一台 DFA，如下图所示，它与原始的 NFA 接受同样的字符串：

注 8：模拟一个三状态的 NFA 时，最差情况是“1”“2”“3”“1 或者 2”“1 或者 3”“2 或者 3”“1、2 或者 3”和“无”。





这个 DFA 只比我们开始的 NFA 多出一个状态，而且对于一些 NFA，这个过程可能会产生比原始机器的状态更少的 DFA。但是在最坏情况下，一台有 n 个状态的 NFA 可能需要一台有 2^n 个状态的 DFA，因为 n 个状态总共有 2^n 个可能组合（考虑把每个组合都表示成一个 n 比特的数字，其中第 n 个比特表示状态 n 是否包含在这个组合中），并且模拟可能需要访问其中所有的组合而不仅仅是其中一部分。

下面我们用 Ruby 实现这个 NFA 到 DFA 的转换。策略是引入一个新的类 `NFASimulation`，用来收集 NFA 模拟的信息然后把这些信息汇总成一台 DFA 。`NFASimulation` 根据特定的 `NFADesign` 创建，并且最后提供一个 `#to_dfa_design` 方法把它转换成等价的 `DFADesign`。

我们已经有了可以模拟 NFA 的 `NFA` 类，因此 `NFASimulation` 可以创建 `NFA` 的实例，然后操纵这个实例弄清楚对所有可能的输入它们都是如何响应的。在开始写 `NFASimulation` 之前，我们先回到 `NFADesign` 并且给 `NFADesign#to_nfa` 增加一个可选的参数“当前状态”，这样就可以使用任意集合的当前状态构建一台 NFA，而不是只能使用 `NFADesgin` 的起始状态：

```
class NFADesign
  def to_nfa(current_states = Set[start_state])
    NFA.new(current_states, accept_states, rulebook)
  end
end
```

此前，一台 NFA 的模拟只能从它的起始状态开始，但这个新的参数让它可以从其他任何点起步：

```
>> rulebook = NFARulebook.new([
     FARule.new(1, 'a', 1), FARule.new(1, 'a', 2), FARule.new(1, nil, 2),
     FARule.new(2, 'b', 3),
     FARule.new(3, 'b', 1), FARule.new(3, nil, 2)
   ])
=> #<struct NFARulebook rules=[...]>
>> nfa_design = NFADesign.new(1, [3], rulebook)
=> #<struct NFADesign start_state=1, accept_states=[3], rulebook=...>
```

```
>> nfa_design.to_nfa.current_states
=> #<Set: {1, 2}>
>> nfa_design.to_nfa(Set[2]).current_states
=> #<Set: {2}>
>> nfa_design.to_nfa(Set[3]).current_states
=> #<Set: {3, 2}>
```



这个 NFA 类自动把自由移动考虑进来了——可以看到 NFA 从状态 3 开始的时候，无需读取任何输入它就可能处于状态 2 或者 3。因此为了支持自由移动，我们不用做任何特别的事情。

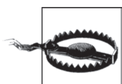
现在我们可以用任何可能状态的集合创建一台 NFA，向其输入一个字符，然后看它最终可能处于什么状态。这是把一台 NFA 转换成一台 DFA 重要的一步。在 NFA 处于状态 2 或者 3 并且读入一个 b 的时候，之后它可能处于什么状态呢？

```
>> nfa = nfa_design.to_nfa(Set[2, 3])
=> #<struct NFA current_states=#<Set: {2, 3}>, accept_states=[3], rulebook=...>
>> nfa.read_character('b'); nfa.current_states
=> #<Set: {3, 1, 2}>
```

答案是状态 1、2 或者 3，就像我们在手工转换过程中发现的那样。（请记住，集合中元素的顺序没关系。）

让我们使用这个思想创建 NFASimulation 类，给它增加一个方法计算模拟的状态如何根据某一个特定的输入而改变。我们把模拟的状态看成这台 NFA 当前可能状态的集合（例如"1、2 或者 3"），因此可以写一个 #next_state 方法，以一个模拟的状态和一个字符为参数，把这个字符传递给对应那个状态的一台 NFA，之后通过监视这台 NFA 得到一个新的状态：

```
class NFASimulation < Struct.new(:nfa_design)
  def next_state(state, character)
    nfa_design.to_nfa(state).tap { |nfa|
      nfa.read_character(character)
    }.current_states
  end
end
```



这里讨论的两种状态很容易让人感到迷惑。模拟的一个状态（NFASimulation#next_state 的 state 参数）是许多 NFA 状态的一个集合，这是为什么我们可以把它作为 NFADesign#to_nfa 的 current_states 参数的原因。

这让我们可以很方便地考察模拟的不同状态：

```
>> simulation = NFASimulation.new(nfa_design)
=> #<struct NFASimulation nfa_design=...>
```

```
>> simulation.next_state(Set[1, 2], 'a')
=> #<Set: {1, 2}>
>> simulation.next_state(Set[1, 2], 'b')
=> #<Set: {3, 2}>
>> simulation.next_state(Set[3, 2], 'b')
=> #<Set: {1, 3, 2}>
>> simulation.next_state(Set[1, 3, 2], 'b')
=> #<Set: {1, 3, 2}>
>> simulation.next_state(Set[1, 3, 2], 'a')
=> #<Set: {1, 2}>
```

现在需要一种方式能系统地考察模拟的状态并把我们的发现记录成一台 DFA 的状态和规则。我们打算直接使用每个模拟的状态作为一个 DFA 状态，因此第一步是实现 `NFASimulation#rules_for`，它使用 `#next_state` 发现每一个规则的目的状态，从一个特定的模拟状态出发构建出全部规则。"全部规则"意味着它是对每一个可能的输入字符适用的一个规则，因此我们还定义了辅助方法 `NFARulebook#alphabet` 来了解原始的 NFA 可以读取哪些字符：

```
class NFARulebook
  def alphabet
    rules.map(&:character).compact.uniq
  end
end

class NFASimulation
  def rules_for(state)
    nfa_design.rulebook.alphabet.map { |character|
      FARule.new(state, character, next_state(state, character))
    }
  end
end
```

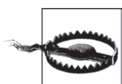
如预期一样，这让我们看到了在不同的状态之间不同的输入将会如何模拟：

```
>> rulebook.alphabet
=> ["a", "b"]
>> simulation.rules_for(Set[1, 2])
=> [
     #<FARule #<Set: {1, 2}> --a--> #<Set: {1, 2}>>,
     #<FARule #<Set: {1, 2}> --b--> #<Set: {3, 2}>>
   ]
>> simulation.rules_for(Set[3, 2])
=> [
     #<FARule #<Set: {3, 2}> --a--> #<Set: {}>>,
     #<FARule #<Set: {3, 2}> --b--> #<Set: {1, 3, 2}>>
   ]
```

方法 `#rules_for` 让我们可以通过已知的模拟状态发现新的状态，并且通过反复对其执行，我们可以找到所有可能的模拟状态。我们可以使用 `NFASimulation#discover_states_and_rules` 方法，它采用类似 `NFARulebook#follow_free_moves` 的方法递归找到更多的状态。

```
class NFASimulation
  def discover_states_and_rules(states)
    rules = states.flat_map { |state| rules_for(state) }
    more_states = rules.map(&:follow).to_set

    if more_states.subset?(states)
      [states, rules]
    else
      discover_states_and_rules(states + more_states)
    end
  end
end
```



#discover_states_and_rules 并不关心模拟状态背后的状态，而只有这个状态才能用作 #rule_for 的参数。但是作为程序员，还有一个地方可能让我们困惑。变量 states 和 more_states 是模拟状态的集合，但是我们知道每一个模拟状态本身是一个 NFA 状态的集合，因此 states 和 more_states 实际上是 NFA 状态集合的集合。

最初，我们只知道模拟的一个状态：NFA 进入起始状态时的可能状态集合。#discover_states_and_rules 从这个起点开始探索，最终找到所有的 4 个状态和模拟的 8 个规则：

```
>> start_state = nfa_design.to_nfa.current_states
=> #<Set: {1, 2}>
>> simulation.discover_states_and_rules(Set[start_state])
=> [
     #<Set: {
       #<Set: {1, 2}>,
       #<Set: {3, 2}>,
       #<Set: {}>,
       #<Set: {1, 3, 2}>
     }>,
     [
       #<FARule #<Set: {1, 2}> --a--> #<Set: {1, 2}>>,
       #<FARule #<Set: {1, 2}> --b--> #<Set: {3, 2}>>,
       #<FARule #<Set: {3, 2}> --a--> #<Set: {}>>,
       #<FARule #<Set: {3, 2}> --b--> #<Set: {1, 3, 2}>>,
       #<FARule #<Set: {}> --a--> #<Set: {}>>,
       #<FARule #<Set: {}> --b--> #<Set: {}>>,
       #<FARule #<Set: {1, 3, 2}> --a--> #<Set: {1, 2}>>,
       #<FARule #<Set: {1, 3, 2}> --b--> #<Set: {1, 3, 2}>>
     ]
   ]
```

最后我们要知道的是，每一个模拟状态是否应该被处理成一个接受状态，但是在模拟中很容易通过查询 NFA 得到结果：

```
>> nfa_design.to_nfa(Set[1, 2]).accepting?
=> false
>> nfa_design.to_nfa(Set[2, 3]).accepting?
```

```
=> true
```

既然我们有了模拟 DFA 的所有部件，现在只需一个 NFASimulation#to_dfa_design 方法把它们封装成一个 DFADesign 实例：

```
class NFASimulation
  def to_dfa_design
    start_state   = nfa_design.to_nfa.current_states
    states, rules = discover_states_and_rules(Set[start_state])
    accept_states = states.select { |state| nfa_design.to_nfa(state).accepting? }

    DFADesign.new(start_state, accept_states, DFARulebook.new(rules))
  end
end
```

就这样。我们可以使用任何 NFA 构造一个 NFASimulation 实例，并把它转换成一个接受同样字符串的 DFA：

```
>> dfa_design = simulation.to_dfa_design
=> #<struct DFADesign ...>
>> dfa_design.accepts?('aaa')
=> false
>> dfa_design.accepts?('aab')
=> true
>> dfa_design.accepts?('bbbabb')
=> true
```

棒极了！

在本节的开始，我们问过 NFA 的额外特性是否能做一台 DFA 完成不了的事情。现在很明显答案为否，因为如果任何 NFA 都可以转成一台做同样工作的 DFA，那么 NFA 就不会有额外的能力。非确定性和自由移动只是一台 DFA 已经能做的工作的再包装，就像编程语言里中的语法糖一样，它们不是让我们超越确定性约束的新能力。

理论上说，为一台简单的机器增加更多的特性却没有为它根本上增加更多的能力非常有趣，但实际上这是很有用的，因为一台 DFA 比一台 NFA 更容易模拟：只有一个当前状态要跟踪，并且一台 DFA 用硬件或者机器代码实现起来足够简单，可以使用程序存储位置作为状态，用条件分支作为规则。这意味着一个正则表达式的实现可以把一个模式先转换成一台 NFA 然后再转换成一台 DFA，得到一台能被快速高效模拟的非常简单的机器。

DFA 最小化

一些 DFA 的特性是**最小化**的，就是说无法设计出一台能接受同样字符串但是状态更少的 DFA。NFA 到 DFA 的转换过程有时候会产生包含冗余状态的非最小化 DFA，但是有一种优雅的方式可以去除这种冗余，叫作 **Brzozowski 算法**。

(1) 从你的非最小化 DFA 开始。

(2) 反转所有规则。从形象的表示上说，这意味着表示机器的图上每一个箭头都保持原位但是方向反转；从代码上说，每一个 `FARule.new(state, character, next_state)` 被替换成 `FARule.new(next_state,character, state)`。反转规则通常会打破确定性约束，因此现在你有了一台 NFA。

(3) 交换起始状态和接受状态的角色：起始状态成为接受状态，而每一个接受状态成为一个起始状态。（因为一台 NFA 只有一个起始状态，所以你不能直接把所有的接受状态变成起始状态，但是你可以创建一个新的起始状态，然后通过自由移动把它与每一个旧的接受状态连接起来，这样效果是一样的。）

(4) 把这个反转的 NFA 按通常方式转换成一台 DFA。

奇怪的是，这样得到的 DFA 保证是最小的而且不含冗余状态。遗憾的缺点是它只能接受原始 DFA 字符串的**颠倒**版本：如果我们原始的 DFA 接受字符串 `'ab'`、`'aab'`、`'aaab'` 等，那这个最小化的 DFA 将接受 `'ba'`、`'baa'` 和 `'baaa'` 形式的字符串。修正方法是简单地第二次执行整个过程，从反转的 DFA 开始再得到一个二次反转的 DFA，它还能保证是最小的，但这次能接受与我们开始的那台机器一样的字符串了。

能有一种自动的方法去除设计中的冗余是很美好的。但有趣的是，一台最小化的 DFA 也是**标准的**：接受完全相同字符串的任何两台 DFA 将最小化成为同样的机器，因此我们可以把两台 NFA 最小化然后比较结果看它们结构是否相同，以此来检查两台 DFA 是否等价。[9] 这反过来提供了一种优雅的方法，可以检查两个正则表达式是否等价：如果我们把与同一个字符串匹配的两个模式（例如 `ab(ab)*` 和 `a(ba)*b`）转换成 NFA，把这些 NFA 转成 DFA，然后把两台 DFA 使用 Brzozowski 算法最小化，最终将得到两台看起来一样的机器。

注 9：解决这个图的同构问题本身要求一个聪明的算法，但非正式地检查两台机器的结构图并确定它们是否“相同”却足够简单。

第 4 章

增加计算能力

第 3 章探讨了有限自动机，这是一种假想的机器，它去掉了真实计算机的复杂性并把其规约成了最简单的形式。我们详细考察了这些机器的行为并了解了它们的用处，而且还发现，非确定性有限自动机虽然有一些奇特的执行方法，但计算能力并不比确定性有限自动机强。

我们没法通过为有限自动机增加非确定性和自由移动这种奇特的特性来提高它的计算能力。这个事实表明，我们已经停留在这些简单机器的计算水平上无法前进了。而且如果不从根本上改变机器的工作方式，将无法脱离这种停滞不前的境地。那么，所有这些机器到底有多强的能力呢？好吧，没有多少能力。它们被限制在非常有限的应用上（只能接受或者拒绝字符序列），而且即使在这么小的范围内，仍然很容易碰到机器无法识别的语言。

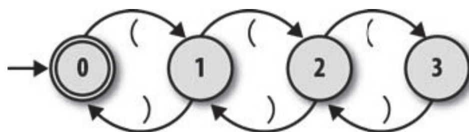
举个例子，假设要设计一台有限自动机，要求它能读取带有左右括号的字符串，并且只有字符串中的左右括号是平衡的（即每一个右括号都能在字符串中找到与其匹配的左括号），它才会接受。[1]

解决这个问题的一般策略是一次读取一个字符，同时跟踪一个表示当前*嵌套级别*的数字：读入一个左括号时增加嵌套级别，读入一个右括号时降低嵌套级别。只要嵌套级别到零了，就表示当前读到的这些括号已经都匹配上了（因为嵌套级别增加和减少的数量是一样的），并且如果我们试图把嵌套级别降低到小于零的值，那就表明当前的右括号多了

注 1：这与接受仅包含同样数量的左右括号的字符串完全不同。字符串 '()' 和 ')(' 都有一个左括号和一个右括号，但只有 '()' 是平衡的。

（如 '())'），不管还有什么字符没有读取，字符串里的括号一定已经不平衡了。

作为一个良好的开始，我们可以为这个任务设计一台 NFA。下面是拥有四个状态的 NFA：



每个状态都对应一个嵌套级别，读取一个左括号或者一个右括号会分别让机器转移到与更高或者更低级别对应的状态，“没有嵌套”对应的就是接受状态。我们已经实现了用 Ruby 模拟这台 NFA 所需要的一切，因此来运行一下：

```
>> rulebook = NFARulebook.new([
     FARule.new(0, '(', 1), FARule.new(1, ')', 0),
     FARule.new(1, '(', 2), FARule.new(2, ')', 1),
     FARule.new(2, '(', 3), FARule.new(3, ')', 2)
   ])
=> #<struct NFARulebook rules=[...]>
>> nfa_design = NFADesign.new(0, [0], rulebook)
=> #<struct NFADesign start_state=0, accept_states=[0], rulebook=...>
```

对于某些输入，我们的 NFA 工作得很好。它能确定 '(()' 和 '())' 的括号不平衡，而 '(())' 的括号是平衡的，它甚至能识别 '(()(()()))' 这种更为复杂的平衡字符串：

```
>> nfa_design.accepts?('(()')
=> false
>> nfa_design.accepts?('())')
=> false
>> nfa_design.accepts?('(())')
=> true
>> nfa_design.accepts?('(()(()()))')
=> true
```

可是这种设计有一个严重的缺陷：如果括号的嵌套等级超过 3，它就会失败。它没有足够多的状态跟踪 '(((())))' 这样的字符串的嵌套，因此即使括号明显是平衡的它也会拒绝：

```
>> nfa_design.accepts?('(((())))')
=> false
```

我们可以通过临时增加更多的状态来修正此问题。一台拥有 5 个状态的 NFA 可以识别任意嵌套级别小于 5 的平衡字符串，而一台拥有 10 个、100 个或者 1000 个状态的 NFA，可以识别嵌套级别在机器硬限制以内的任意平衡字符串。但是，我们如何设计支持任意嵌套级别、能识别任意平衡字符串的 NFA 呢？结论是设计不出来：一台有限自动机的状态数总是有限的，因此任何机器能支持的嵌套级别也总是有限的，我们只要提供一个比它能处理的嵌套级别多一级的字符串，它就无法处理了。

根本问题是一台有限自动机只有固定的状态集合，因而其存储是有限的，因此没法跟踪任意数量的信息。在平衡字符串问题当中，一台 NFA 很容易递增到设计时限制的某个最大数目，但无法继续计数以适应任何可能大小的输入。[2] 本质上大小固定的任务（比如对字符串 'abc' 进行匹配），或者无需跟踪重复次数的任务（比如对正则表达式 ab*c 进行匹配），都不受这个问题的影响，但在信息数目不可预知，需要在计算过程中存储并在之后重用的场景下，这个问题会让有限自动机无能为力。

正则表达式和嵌套字符串

我们已经看到，有限自动机与正则表达式关系密切。3.3.2 节展示了如何把任意一个正则表达式转换成一台 NFA，并且实际上还有一个算法可以把任意 NFA 转换回一个正则表达式。[3] 这告诉我们正则表达式与 NFA 等价并且拥有同样的限制，因此也不可能使用正则表达式识别括号组成的平衡字符串，也不能识别所有定义中牵涉嵌套任意深度配对情况的语言。

关于这个缺点，最知名的例子就是正则表达式无法区分有效 HTML 和无效 HTML（http://stackoverflow.com/a/1732454）这一事实。许多 HTML 元素要求开闭标记成对出现，而这些标记自身还可能封装着其他元素，因此有限自动机没有足够的能力读取 HTML 字符串，并同时跟踪哪些标记没有配上对以及它们嵌套的深度是多少。

但实际上，现实世界中的“正则表达式”库经常超越正则表达式理论上所拥有的能力。Ruby 的 Regexp 对象提供的很多特性都不在正则表达式的形式定义当中，而且这些特性提供的额外能力可以识别更多语言。

Regexp 加强的一点就是可以把一个子表达式用 (?<*name*>) 语法标记，然后在别的地方使用 \g<*name*>“调用”这个子表达式。能够引用自己的子表达式，这使得一个 Regexp 能够递归调用自身，这让匹配任意深度的成对嵌套成为可能。

例如，尽管 NFA 不能匹配括号的平衡字符串（因此理论上说正则表达式也不能），但子表达式调用允许我们写出匹配这种字符串的 Regxp。下面就是这个 Regxp 的样子：

```
balanced =
  /
    \A              # 匹配开始于字符串的开头
    (?<brackets>    # 叫作 "brackets" 的子表达式开始
      \(            # 匹配左括号
      \g<brackets>* # 匹配子表达式 "brackets" 零次或者多次
      \)            # 匹配右括号
```

注 2：这并不是说一个输入字符串真的可以是无限的，只是说我们可以根据需要让它尽可能有限地大。

注 3：简单地说，这个算法通过把一台 NFA 转换成广义非确定性有限自动机（GNFA）来完成工作。GNFA 是这样一种有限状态机，每一个规则都用一个正则表达式标记（而不是用一个字符标记），然后不断合并这台 GNFA 的状态和规则，直到只剩下两个状态和一个规则为止。最后剩下的规则上标记的正则表达式总是与原始 NFA 匹配相同的字符串。

```
      )               # 子表达式结束
      *               # 重复整个模式零次或多次
      \z              # 匹配结束于字符串的结尾
    /x
```

子表达式 (?<brackets>...) 匹配一对开闭括号，但在括号内，它还能匹配任意次数的自身，因此整个模式可以正确识别嵌套任意深度的括号：

```
>> ['(()', '())', '(())', '(()(()()))', '((((((((((()))))))))))'].grep(balanced)
=> ["(())", "(()(()()))", "((((((((((()))))))))))"]
```

这种方式能行，只是因为 Ruby 的正则表达式引擎使用了**调用栈**跟踪 (?<brackets>...)，这是 DFA 和 NFA 不能做到的。下一节里，我们将看到如何扩展有限自动机，让它也获得这种能力。

是的，你也可以用同样的思想写一个 Regxp 匹配嵌套的 HTML 标记，但肯定不值得花这个时间。

很明显这些机器的能力存在局限性。如果非确定性不足以让有限自动机能力更强，那什么才能赋予它更多的能力呢？现在的问题来源是机器有限的存储，因此我们可以增加一些存储看看怎么样。

4.1 确定性下推自动机

为了解决存储问题，我们可以使用专门的原始空间扩展有限状态自动机，它负责在计算过程中存储数据。除状态提供的有限内部存储之外，这个空间给了机器一种外部存储（external memory）。就像我们将会发现的那样，拥有外部存储对于一台机器的计算能力关系重大。

4.1.1 存储

为有限自动机增加存储的简单方式就是让它可以访问栈，这是一个后进先出的数据结构，可以把字符推入和弹出。栈是简单而且有限制的数据结构——在任意时间都只有顶端的字符可以访问。为了查明栈下面位置的数据，我们只能丢弃顶层的字符，而一旦向栈内推入一串字符，我们就只能按相反的顺序把它们弹出——但它确实可以很好地解决有限存储的问题。对于栈的大小并没有内在的限制，因此原则上它可以根据需要存储数据。[4]

自带栈的有限状态机叫作下推自动机（PushDown Automaton，PDA），如果这台机器的规则是确定性的，我们就叫它确定性下推自动机（Deterministic PushDown Automaton，

注 4：当然，栈在现实世界中的任何实现都受限于计算机的 RAM，或者硬盘上的空闲空间，或者宇宙中原子的数量，但是对于思维实验，我们将认为这些约束都不存在。

DPDA）。能对栈进行访问带来了新的可能性，例如，很容易设计一台 DPDA 来识别括号组成的平衡字符串。下面是它的工作方式。

- 给机器两个状态：1 和 2，状态 1 作为接受状态。
- 状态 1 作为机器的起始状态，此时栈为空。
- 如果处于状态 1 并且读入一个左括号，就把某个字符——我们使用 b 表示“括号”——入栈，并转移到状态 2。
- 如果处于状态 2 并且读入一个左括号，就把字符 b 入栈。
- 如果处于状态 2 并且读入一个右括号，就把字符 b 从栈中弹出。
- 如果处于状态 2 且栈为空，就转移回状态 1。

这台 DPDA 使用栈的大小来记录到目前为止没有配上对的左括号数目。栈为空时，意味着每一个左括号都已经匹配上了右括号，因此字符串一定是平衡的。我们观察一下机器读入字符串 '(()(()()))' 时栈的增长和缩减情况：

状态	是否接受	栈的内容	剩余输入	动作
1	是		(()(()()))	读入 (，推入 b，转移到状态 2
2	否	b	()(()()))	读入 (，推入 b
2	否	bb	)(()()))	读入)，弹出 b
2	否	b	(()()))	读入 (，推入 b
2	否	bb	()()))	读入 (，推入 b
2	否	bbb	)()))	读入)，弹出 b
2	否	bb	()))	读入 (，推入 b
2	否	bbb	)))	读入)，弹出 b
2	否	bb	))	读入)，弹出 b
2	否	b	)	读入)，弹出 b
2	否			转移到状态 1
1	是			——

4.1.2 规则

括号平衡问题 DPDA 背后的思想非常简单，但在我们实际构建它之前，需要弄清楚一些技术细节。首先，我们必须确定下推自动机的工作规则。这里有几个设计问题。

- 每个规则都要修改栈，或者读取输入，或者改变状态，还是三者都要做？
- 推入和弹出需要两种不同的规则吗？
- 栈为空时，我们是否需要一种特殊的规则改变状态呢？
- 就像 NFA 中的自由移动那样，没有从输入读取就改变状态是否可以呢？
- 如果一台 DPDA 可以自发改变状态，那“确定性”是什么意思呢？

通过选择一种足够灵活、能满足所有要求的规则类型，我们可以回答全部问题。我们把一

个 PDA 规则分成 5 部分：

- 机器的当前状态；
- 必须从输入读取的字符（可选）；
- 机器的下一个状态；
- 必须从栈中弹出的字符；
- 栈顶字符弹出后需要推入栈中的字符序列。

前三部分很熟悉，它们来自 DFA 和 NFA 的规则。如果一个规则不想让机器改变状态，它可以让下一个状态与当前状态一样；如果它不想读取任何输入（也就是自由移动），则可以忽略输入字符，只要这不让机器变成非确定性的就可以（参见 4.1.3 节）。

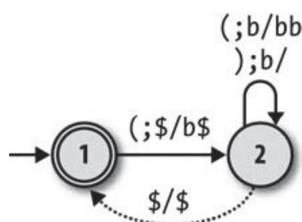
其他两部分——要弹出的字符和要推入的字符序列——是 PDA 特有的。假定一台 PDA 总是要弹出栈顶字符，然后向栈中推入其他字符。每一个规则声明它想要弹出哪个字符，然后这个规则只会在这个字符处于栈顶位置时才会应用；如果这个规则想让那个字符留在栈中而不弹出，它可以把这个字符包含在后来要推入栈中的字符序列当中。

这个五部分的规则格式没有说明栈为空时如何写规则，但我们可以通过选择一个特殊符号标记栈底位置来解决——流行的选择是 `$`——然后每当想要检测栈是否为空时，检查这个符号就可以了。使用这个约定时，最重要的是永远不要让栈真的变空，因为在栈顶为空时没有规则可以应用。机器开始的时候这个特殊的栈底符号应该已经在栈中，任何规则在把这个符号弹出之后必须再次把它推入。

很容易用这种格式重写平衡括号的 DPDA 规则：

- 处于状态 1 而且读入左括号时，弹出字符 `$`，推入字符 `b$`，然后转移到状态 2；
- 处于状态 2 而且读入左括号时，弹出字符 `b`，推入字符 `bb`，然后保持在状态 2；
- 处于状态 2 而且读入右括号时，弹出字符 `b`，不推入任何字符，然后保持在状态 2；
- 处于状态 2（没有读入任何字符）时，弹出字符 `$`，推入字符 `$`，然后转移到状态 1。

我们可以用这个机器的图来展示这些规则。DPDA 图看起来与 NFA 图很像，但 DPDA 图的每个箭头不仅要标记它从输入读取的字符，还要标记这个规则需要弹出和推入的字符。如果我们使用符号 `a;b/cd` 来标记一个规则，它表明从输入读取 `a`，从栈中弹出 `b`，然后向栈中推入 `cd`，这个机器看起来像是这样：



4.1.3 确定性

下一个难题就是为 PDA 准确地定义确定性的含义。对于 DFA 来说，我们的约束是“不能存在冲突”：不能在任何状态上，由于冲突的规则而使机器的下一次移动有二义性。这也适用于 DPDA，例如，在机器处于状态 2、下一个输入字符是左括号并且栈顶是 b 的时候，我们只能应用一个规则。甚至写一个不读取任何输入的自由移动规则都是可以的，只要对于同样的状态和同样的栈顶字符没有其他规则可用就可以，因为这样在确定一个字符是否应该从输入读取的时候会产生二义性。

DFA 还有“不能有遗漏”的约束（每一个可能的情况都应该有一个规则），但是因为状态、输入字符和栈顶字符有大量可能的组合，所以这对于 DPDA 来说很难处理。通常只是忽略这个约束并允许 DPDA 只定义完成工作所需的规则，并且假定一台 DPDA 在没有规则可用时将进入停滞状态。我们的平衡括号 DPDA 在读取 ')' 或 '())' 这样的字符串时会进入这种情况，因为处于状态 1 且读入一个右括号时没有规则可用。

4.1.4 模拟

既然处理完了技术细节，让我们构建一个确定性下推自动机的 Ruby 模拟吧，这样就可以与它交互了。在模拟 DFA 和 NFA 的时候我们已经完成了大部分困难的工作，因此这次只需要微调。

我们缺少的最重要的东西是栈。下面是一种实现栈类的方式：

```
class Stack < Struct.new(:contents)
  def push(character)
    Stack.new([character] + contents)
  end

  def pop
    Stack.new(contents.drop(1))
  end

  def top
    contents.first
  end

  def inspect
    "#<Stack (#{top})#{contents.drop(1).join}>"
  end
end
```

一个 Stack 对象把它的内容存储在一个数组内并把简单的 #push 和 #pop 操作暴露出来以支持字符的推入和弹出，另外还有一个 #top 操作可以读取栈顶的字符：

```
>> stack = Stack.new(['a', 'b', 'c', 'd', 'e'])
=> #<Stack (a)bcde>
```

```
>> stack.top
=> "a"
>> stack.pop.pop.top
=> "c"
>> stack.push('x').push('y').top
=> "y"
>> stack.push('x').push('y').pop.top
=> "x"
```



这仅仅是一个纯功能性的栈。#push 和 #pop 方法是非破坏性的：它们每一个都返回一个新的栈实例而不是修改已有的栈。每次都创建一个新的栈对象比通常拥有破坏性 #push 和 #pop 方法操作的栈（如果我们想这样，可以直接使用 Array）效率要低，但是使用起来要更容易，因为在多处使用一个 Stack 对象的时候，我们不必担心对其进行修改的后果。

第 3 章里，我们可以通过只跟踪一条信息来模拟确定性有限自动机，也就是跟踪 DFA 的当前状态，然后在每次从输入读取字符时使用规则手册更新该状态。但是关于下推自动机计算的每一步有两件重要的事情要知道：它的当前状态是什么，栈的当前内容是什么。如果我们使用名词配置表示一个状态和一个栈的组合，则在下推自动机读取输入字符时，我们可以说它从一个配置转移到了另一个配置，这比总是需要分别提到状态和栈要容易。从这个角度看的话，一台 DPDA 只会有一个当前配置，并且每次读取一个字符时规则手册都会告诉我们如何把当前配置转换成下一个配置。

下面是用来存储 PDA 配置（一个状态和一个栈）的一个 PDAConfiguration 类，以及一个用来表示一台 PDA 的规则手册中的一个规则的 PDARule 类：[5]

```
class PDAConfiguration < Struct.new(:state, :stack)
end

class PDARule < Struct.new(:state, :character, :next_state,
                           :pop_character, :push_characters)
  def applies_to?(configuration, character)
    self.state == configuration.state &&
      self.pop_character == configuration.stack.top &&
      self.character == character
  end
end
```

只有在机器状态、栈顶字符和下一个输入的字符都为期望值的时候才能应用规则：

```
>> rule = PDARule.new(1, '(', 2, '$', ['b', '$'])
=> #<struct PDARule
    state=1,
```

注 5：因为它们的实现并没有做任何确定性的假设，所以这些类的名字以 PDA 开头，而不是以 DPDA 开头，这样它们在模拟非确定性 PDA 时也工作得很好。

```
    character="(",
    next_state=2,
    pop_character="$",
    push_characters=["b", "$"]
  >
>> configuration = PDAConfiguration.new(1, Stack.new(['$']))
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
>> rule.applies_to?(configuration, '(')
=> true
```

对一台有限自动机来说，遵守规则只是意味着从一个状态变成另一个状态，但一个 PDA 规则除了改变状态之外还会更新栈的内容，因此 PDARule#follow 需要接受机器的当前配置作为参数然后返回下一个配置：

```
class PDARule
  def follow(configuration)
    PDAConfiguration.new(next_state, next_stack(configuration))
  end

  def next_stack(configuration)
    popped_stack = configuration.stack.pop

    push_characters.reverse.
      inject(popped_stack) { |stack, character| stack.push(character) }
  end
end
```



如果我们把一些字符先推入栈中然后再把它们弹出，则它们出来时的顺序会与之前的顺序相反：

```
>> stack = Stack.new(['$']).push('x').push('y').push('z')
=> #<Stack (z)yx$>
>> stack.top
=> "z"
>> stack = stack.pop; stack.top
=> "y"
>> stack = stack.pop; stack.top
=> "x"
```

PDARule#next_stack 通过在把字符推入栈之前先把 push_characters 反转的办法解决这个问题。例如，push_characters 的最后一个字符实际上是推入栈中的第一个字符，这样再次弹出的时候它就又是最后一个字符了。这只是为了方便我们把规则的 push_characters 按照字符序列读取（以“弹出的顺序”），这些字符序列在规则应用之后会出现在栈顶，这样我们就不用关心它们到达栈顶的机制了。

因此，如果把一个 PDARule 应用到一个 PDAConfiguration 上，就可以通过这个规则找出接下来的状态和栈是什么样的：

```
>> rule.follow(configuration)
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

这足以实现 DPDA 的规则手册了。这个实现与 3.1.4 节的 DFARulebook 类似：

```
class DPDARulebook < Struct.new(:rules)
  def next_configuration(configuration, character)
    rule_for(configuration, character).follow(configuration)
  end

  def rule_for(configuration, character)
    rules.detect { |rule| rule.applies_to?(configuration, character) }
  end
end
```

现在我们可以为平衡括号 DPDA 汇编一个规则手册了，然后尝试手工单步调试一些配置和输入字符：

```
>> rulebook = DPDARulebook.new([
     PDARule.new(1, '(', 2, '$', ['b', '$']),
     PDARule.new(2, '(', 2, 'b', ['b', 'b']),
     PDARule.new(2, ')', 2, 'b', []),
     PDARule.new(2, nil, 1, '$', ['$'])
   ])
=> #<struct DPDARulebook rules=[...]>
>> configuration = rulebook.next_configuration(configuration, '(')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
>> configuration = rulebook.next_configuration(configuration, '(')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)b$>>
>> configuration = rulebook.next_configuration(configuration, ')')
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

为了代替手工操作，我们可以使用规则手册构建一个 DPDA 对象，它会在从输入读取字符的同时跟踪机器的当前配置：

```
class DPDA < Struct.new(:current_configuration, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_configuration.state)
  end

  def read_character(character)
    self.current_configuration =
      rulebook.next_configuration(current_configuration, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end
end
```

这样我们可以创建一个 DPDA，提供输入，然后看它是否能够接受这些输入：

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA ...>
>> dpda.accepting?
=> true
>> dpda.read_string('(()'); dpda.accepting?
=> false
>> dpda.current_configuration
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)$>>
```

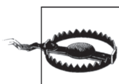
到目前为止一切都很好，但我们正在使用的规则手册中包含一个自由移动，因此模拟需要支持自由移动以便正确工作。让我们增加一个 DPDARulebook 的辅助方法以处理自由移动，这与 NFARulebook 中的类似（参见 3.2.2 节）：

```
class DPDARulebook
  def applies_to?(configuration, character)
    !rule_for(configuration, character).nil?
  end

  def follow_free_moves(configuration)
    if applies_to?(configuration, nil)
      follow_free_moves(next_configuration(configuration, nil))
    else
      configuration
    end
  end
end
```

DPDARulebook#follow_free_moves 将不断地反复执行能应用到当前配置的任何自由移动，直到没有自由移动的时候才会停止：

```
>> configuration = PDAConfiguration.new(2, Stack.new(['$']))
=> #<struct PDAConfiguration state=2, stack=#<Stack ($)>>
>> rulebook.follow_free_moves(configuration)
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
```



在我们的状态机实验中，这是首次在模拟中引入了有可能的无限循环。只要有一个自由移动链，且它的开始和结束状态相同，就会有循环。最简单的例子是存在一个根本不改变配置的自由移动：

```
>> DPDARulebook.new([PDARule.new(1, nil, 1, '$', ['$'])]).
     follow_free_moves(PDAConfiguration.new(1, Stack.new(['$'])))
SystemStackError: stack level too deep
```

这些无限循环毫无用处，因此我们在设计下推自动机的时候要注意避免它们。

我们还需要封装 DPDA#current_configuration 的默认实现，以便利用规则手册对自由移动的支持：

```
class DPDA
  def current_configuration
    rulebook.follow_free_moves(super)
  end
end
```

现在我们有了可以启动、接受字符输入并且检查是否接受输入的 DPDA 模拟了：

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA ...>
>> dpda.read_string('(()('); dpda.accepting?
=> false
>> dpda.current_configuration
=> #<struct PDAConfiguration state=2, stack=#<Stack (b)b$>>
>> dpda.read_string('))()'); dpda.accepting?
=> true
>> dpda.current_configuration
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
```

如果把此模拟像往常一样封装进 DPDADesign，我们就可以很容易地根据需要检查字符串：

```
class DPDADesign < Struct.new(:start_state, :bottom_character,
                              :accept_states, :rulebook)
  def accepts?(string)
    to_dpda.tap { |dpda| dpda.read_string(string) }.accepting?
  end

  def to_dpda
    start_stack = Stack.new([bottom_character])
    start_configuration = PDAConfiguration.new(start_state, start_stack)
    DPDA.new(start_configuration, accept_states, rulebook)
  end
end
```

不出所料，我们的 DPDA 可以识别任意嵌套深度的平衡括号组成的复杂字符串：

```
>> dpda_design = DPDADesign.new(1, '$', [1], rulebook)
=> #<struct DPDADesign ...>
>> dpda_design.accepts?('(((((((((())))))))))')
=> true
>> dpda_design.accepts?('()(())((()))(()(()))')
=> true
>> dpda_design.accepts?('(()(()(()()(()()))()')
=> false
```

还有最后一个细节要注意。输入后 DPDA 处于有效状态时，我们的模拟运行得很完美，但在机器卡住的时候它就会出问题了：

```
>> dpda_design.accepts?('())')
NoMethodError: undefined method `follow' for nil:NilClass
```

之所以会发生这种情况，是因为 DPDARulebook#next_configuration 假设它总能找到可用的

规则，因此在没有规则可用的时候我们不应该调用它。修改 `DPDA#read_character` 检查可用规则，如果没有可用规则，就把 DPDA 置于一个无法转移出去的阻塞状态，这样我们就解决了这个问题：

```
class PDAConfiguration
  STUCK_STATE = Object.new

  def stuck
    PDAConfiguration.new(STUCK_STATE, stack)
  end

  def stuck?
    state == STUCK_STATE
  end
end

class DPDA
  def next_configuration(character)
    if rulebook.applies_to?(current_configuration, character)
      rulebook.next_configuration(current_configuration, character)
    else
      current_configuration.stuck
    end
  end

  def stuck?
    current_configuration.stuck?
  end

  def read_character(character)
    self.current_configuration = (next_configuration(character))
  end

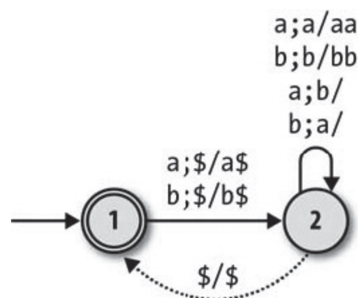
  def read_string(string)
    string.chars.each do |character|
      read_character(character) unless stuck?
    end
  end
end
```

现在 DPDA 会优雅地阻塞住而不会崩溃了：

```
>> dpda = DPDA.new(PDAConfiguration.new(1, Stack.new(['$'])), [1], rulebook)
=> #<struct DPDA ...>
>> dpda.read_string('())'); dpda.current_configuration
=> #<struct PDAConfiguration state=#<Object>, stack=#<Stack ($)>>
>> dpda.accepting?
=> false
>> dpda.stuck?
=> true
>> dpda_design.accepts?('())')
=> false
```

4.2 非确定性下推自动机

尽管处理平衡括号问题的机器确实需要栈来完成工作，但它其实只是将栈作为一个计数器，并且它的规则只区分“栈为空”和“栈不为空”。更复杂的 DPDA 将会把一种以上的符号推入栈中，并在执行计算时使用这些信息。一个简单的例子是一台机器，它能识别包含相等数目的两种字符的字符串，比如 a 和 b：

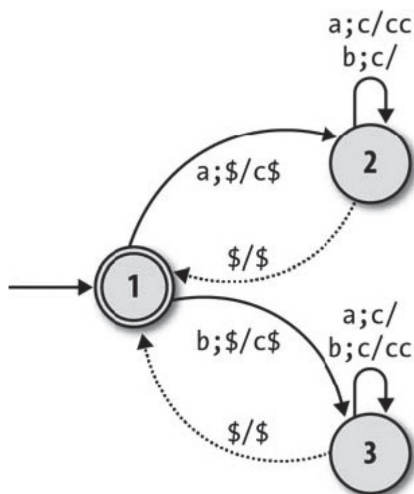


我们的模拟表明它能完成工作：

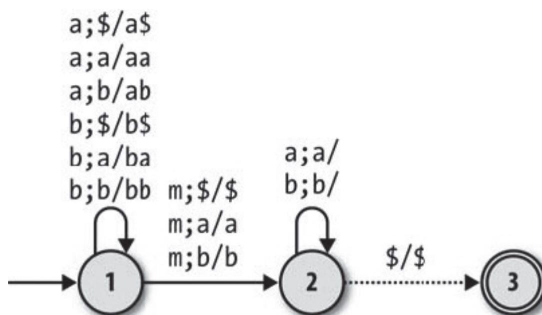
```
>> rulebook = DPDARulebook.new([
     PDARule.new(1, 'a', 2, '$', ['a', '$']),
     PDARule.new(1, 'b', 2, '$', ['b', '$']),
     PDARule.new(2, 'a', 2, 'a', ['a', 'a']),
     PDARule.new(2, 'b', 2, 'b', ['b', 'b']),
     PDARule.new(2, 'a', 2, 'b', []),
     PDARule.new(2, 'b', 2, 'a', []),
     PDARule.new(2, nil, 1, '$', ['$'])
   ])
=> #<struct DPDARulebook rules=[...]>
>> dpda_design = DPDADesign.new(1, '$', [1], rulebook)
=> #<struct DPDADesign ...>
>> dpda_design.accepts?('ababab')
=> true
>> dpda_design.accepts?('bbbaaaab')
=> true
>> dpda_design.accepts?('baa')
=> false
```

这与平衡括号的机器类似，只是它的行为由栈顶字符控制。a 在栈顶意味着机器已经看到 a 过剩了，因此任何额外从输入读取的 a 将会在栈中累积，而每读到一个 b 就会从栈中弹出一个 a 作为抵销；反之，栈顶是 b 时，就是 b 在累积而用 a 来抵销。

即使是这个 DPDA 也没有利用栈的全部优点。在栈顶字符之下没有它感兴趣的任何历史数据，只有一些无意义的 a 或 b，因此我们可以只把一种字符推入栈（也就是说还是把它当作一个简单的计数器），并使用两个不同的状态区分“对过剩的 a 计数”和“对过剩的 b 计数”，这样也能得到同样的结果：



为了真正开发出栈的潜能，我们需要一个更难的问题强迫我们存储结构化信息。经典的例子是识别回文字符串：随着一个字符一个字符地读取输入字符串，我们需要记住所看到的数据；一旦字符串读取过了一半，就要检查内存以确定之前看到的字符是否为当前呈现字符的逆序。下面这个 DPDA 能够识别一个回文字符串，这个字符串由字符 a 和 b 组成，并且在中间的位置有一个字符 m（表示中间位置）：



这台机器从状态 1 开始，不断从输入读取 a 和 b，然后把它们推入栈中。它读到 m 的时候，会转移到状态 2，在那里一直读取输入字符同时尝试把每一个字符都弹出栈。如果字符串后半部分的每一个字符都与栈中弹出的内容匹配，机器就停留在状态 2 并最终碰到栈底的 $，此时转移到状态 3 并接受这个输入字符串。处于状态 2 的时候，如果读入的任何字符与栈顶的字符不匹配，那就没有规则可以遵守，因此它将进入阻塞状态并拒绝这个字符串。

我们可以模拟这台 DPDA 检查它的工作情况：

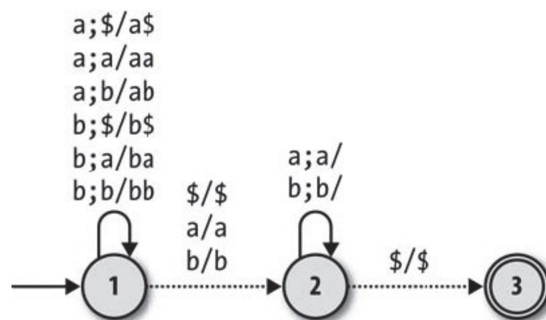
```
>> rulebook = DPDARulebook.new([
     PDARule.new(1, 'a', 1, '$', ['a', '$']),
     PDARule.new(1, 'a', 1, 'a', ['a', 'a']),
```

```
    PDARule.new(1, 'a', 1, 'b', ['a', 'b']),
    PDARule.new(1, 'b', 1, '$', ['b', '$']),
    PDARule.new(1, 'b', 1, 'a', ['b', 'a']),
    PDARule.new(1, 'b', 1, 'b', ['b', 'b']),
    PDARule.new(1, 'm', 2, '$', ['$']),
    PDARule.new(1, 'm', 2, 'a', ['a']),
    PDARule.new(1, 'm', 2, 'b', ['b']),
    PDARule.new(2, 'a', 2, 'a', []),
    PDARule.new(2, 'b', 2, 'b', []),
    PDARule.new(2, nil, 3, '$', ['$'])
  ])
=> #<struct DPDARulebook rules=[...]>
>> dpda_design = DPDADesign.new(1, '$', [3], rulebook)
=> #<struct DPDADesign ...>
>> dpda_design.accepts?('abmba')
=> true
>> dpda_design.accepts?('babbamabbab')
=> true
>> dpda_design.accepts?('abmb')
=> false
>> dpda_design.accepts?('baambaa')
=> false
```

这很好，但是输入字符串中间的 m 是一种逃避。我们为什么不能设计一台机器，让它能识别回文字符串——aa、abba、babbaabbab 等——但无需在中间插入一个标记呢？

机器在到达字符串的中间位置时需要从状态 1 转移到状态 2，而没有标记的话，就没法知道什么时候做这样的状态转移。就像我们之前处理 NFA 时看到的那样，这种“我怎么知道什么时候该……”的问题可以通过放松确定性约束并允许机器在任意时间都可以做重要的状态转移来解决，这样它就可能通过在正确的时间遵照正确的规则接受一个回文字符串。

不出所料的是，没有确定性约束的下推自动机叫作*非确定性下推自动机*（nondeterministic pushdown automaton）。下面是一台能识别由偶数个字母组成的回文字符串的非确定性下推自动机[6]：



注 6：“偶数个字母”的约束能让机器保持简单：一个长度是 2n 的回文字符串可以通过先把 n 个字符推入栈然后再把 n 个字符弹出栈来接受。为了识别任意的回文字符串，需要从状态 1 到状态 2 之间多一些规则。

除了状态 1 到状态 2 的规则，这和 DPDA 的版本是一样的：在 DPDA 中，它们从输入读取 m，但这里是自由移动。这让 NPDA 有机会在输入字符串的时候改变状态，而不再需要标记了。

4.2.1　模拟

一台非确定性机器要比一台确定性机器更难模拟，但我们在 3.2.1 节中已经完成了 NFA 中困难的部分，因此可以在处理 NPDA 时重用同样的思想。我们需要一个 NPDARulebook 来保存一个 PDARule 的非确定性集合，它的实现也几乎和 NFARulebook 完全一样：

```
require 'set'

class NPDARulebook < Struct.new(:rules)
  def next_configurations(configurations, character)
    configurations.flat_map { |config| follow_rules_for(config, character) }.to_set
  end

  def follow_rules_for(configuration, character)
    rules_for(configuration, character).map { |rule| rule.follow(configuration) }
  end

  def rules_for(configuration, character)
    rules.select { |rule| rule.applies_to?(configuration, character) }
  end
end
```

在 3.2.1 节中，我们通过跟踪可能状态的集合来模拟一台 NFA，这里会通过可能配置的集合来模拟一台 NPDA。

我们的规则手册需要支持自由移动，这又一次几乎与 NFARulebook 的实现一致：

```
class NPDARulebook
  def follow_free_moves(configurations)
    more_configurations = next_configurations(configurations, nil)
    if more_configurations.subset?(configurations)
      configurations
    else
      follow_free_moves(configurations + more_configurations)
    end
  end
end
```

在当前配置的集合之外，我们还需要一个 NPDA 类来封装一个规则手册：

```
class NPDA < Struct.new(:current_configurations, :accept_states, :rulebook)
  def accepting?
    current_configurations.any? { |config| accept_states.include?(config.state) }
  end

  def read_character(character)
```

```
    self.current_configurations =
      rulebook.next_configurations(current_configurations, character)
  end

  def read_string(string)
    string.chars.each do |character|
      read_character(character)
    end
  end

  def current_configurations
    rulebook.follow_free_moves(super)
  end
end
```

这让我们可以随着每个字符的读入单步模拟出所有可能的配置：

```
>> rulebook = NPDARulebook.new([
     PDARule.new(1, 'a', 1, '$', ['a', '$']),
     PDARule.new(1, 'a', 1, 'a', ['a', 'a']),
     PDARule.new(1, 'a', 1, 'b', ['a', 'b']),
     PDARule.new(1, 'b', 1, '$', ['b', '$']),
     PDARule.new(1, 'b', 1, 'a', ['b', 'a']),
     PDARule.new(1, 'b', 1, 'b', ['b', 'b']),
     PDARule.new(1, nil, 2, '$', ['$']),
     PDARule.new(1, nil, 2, 'a', ['a']),
     PDARule.new(1, nil, 2, 'b', ['b']),
     PDARule.new(2, 'a', 2, 'a', []),
     PDARule.new(2, 'b', 2, 'b', []),
     PDARule.new(2, nil, 3, '$', ['$'])
   ])
=> #<struct NPDARulebook rules=[...]>
>> configuration = PDAConfiguration.new(1, Stack.new(['$']))
=> #<struct PDAConfiguration state=1, stack=#<Stack ($)>>
>> npda = NPDA.new(Set[configuration], [3], rulebook)
=> #<struct NPDA ...>
>> npda.accepting?
=> true
>> npda.current_configurations
=> #<Set: {
     #<struct PDAConfiguration state=1, stack=#<Stack ($)>>,
     #<struct PDAConfiguration state=2, stack=#<Stack ($)>>,
     #<struct PDAConfiguration state=3, stack=#<Stack ($)>>
   }>
>> npda.read_string('abb'); npda.accepting?
=> false
>> npda.current_configurations
=> #<Set: {
     #<struct PDAConfiguration state=1, stack=#<Stack (b)ba$>>,
     #<struct PDAConfiguration state=2, stack=#<Stack (a)$>>,
     #<struct PDAConfiguration state=2, stack=#<Stack (b)ba$>>
   }>
>> npda.read_character('a'); npda.accepting?
=> true
```

```
>> npda.current_configurations
=> #<Set: {
     #<struct PDAConfiguration state=1, stack=#<Stack (a)bba$>>,
     #<struct PDAConfiguration state=2, stack=#<Stack ($)>>,
     #<struct PDAConfiguration state=2, stack=#<Stack (a)bba$>>,
     #<struct PDAConfiguration state=3, stack=#<Stack ($)>>
   }>
```

最后用一个 NPDADesign 类直接测试字符串：

```
class NPDADesign < Struct.new(:start_state, :bottom_character,
                              :accept_states, :rulebook)
  def accepts?(string)
    to_npda.tap { |npda| npda.read_string(string) }.accepting?
  end

  def to_npda
    start_stack = Stack.new([bottom_character])
    start_configuration = PDAConfiguration.new(start_state, start_stack)
    NPDA.new(Set[start_configuration], accept_states, rulebook)
  end
end
```

现在可以检查一下 NPDA 是否确实可以识别回文字符串：

```
>> npda_design = NPDADesign.new(1, '$', [3], rulebook)
=> #<struct NPDADesign ...>
>> npda_design.accepts?('abba')
=> true
>> npda_design.accepts?('babbaabbab')
=> true
>> npda_design.accepts?('abb')
=> false
>> npda_design.accepts?('baabaa')
=> false
```

看起来很好啊！非确定性明显已经给了我们确定性机器所不具备的识别语言的能力。

4.2.2　不等价

但是等一等：我们在 3.4 节中看到，没有栈的非确定性机器在能力上与确定性机器是等价的。我们用 Ruby 模拟的 NFA 行为像是一台 DFA（它们都是随着从输入读取字符在有限个“模拟状态”中转移），可以把任意一台 NFA 转换成接受同样字符的 DFA。那么非确定性真的能带给我们额外的能力，还是 Ruby 模拟的 NPDA 只是行为类似 DPDA 呢？是否存在一个算法能把任意的非确定性下推自动机转换成确定性下推自动机呢？

答案是不存在。NFA 到 DFA 的小把戏能成功，是因为我们可以使用一个 DFA 状态表示多个可能的 NFA 状态。为了模拟一台 NFA，我们只需要跟踪现在它可能处于的状态，然

后每次读取一个输入字符就选一个不同的可能状态集合，这样如果给它设定正确的规则，DFA 就可以轻松完成工作。

但这个小把戏不适用于 PDA：我们不能有效地把多重 NPDA 配置表示成一个 DPDA 配置。并不奇怪，问题出在栈的上面。一个 NPDA 模拟需要知道当前能出现在栈顶的所有字符，而且它必须能同时从几个模拟的栈弹出和推入。无法把所有可能的栈组合成一个栈，以便 DPDA 仍能看到所有的栈顶字符并可以单独访问每个可能的栈。我们用 Ruby 写一个程序做所有这些并不难，但是 DPDA 没有足够的能力来处理。

所以不幸的是，我们的 NPDA 模拟的行为并不像一台 DPDA，也不存在 NDPA 到 DPDA 的算法。无标记的回文问题就是这样一个例子，NPDA 能完成这个问题，但 DPDA 不能，因此非确定性下推自动机确实比确定性的能力要强。

4.3　使用下推自动机进行分析

3.3 节展示了如何用非确定性有限自动机实现正则表达式匹配。下推自动机也有一个重要的实际应用：它们能用来解析编程语言。

在 2.6 节中，我们已经看到如何使用 Treetop 为一部分 Simple 语言构建解析器。Treetop 解析器使用*解析表达式语法*来描述被解析语言的完整语法，但这是一个相当现代的思想。更传统的方式是把解析过程分成两个独立的阶段。

- *词法分析*

  读取一个原始字符串然后把它转换成一个单词 token 序列。每一个单词 token 代表程序语法的一个组成部分，例如“变量名”、“左括号”或者“while 关键字”。词法分析器使用称为*词法*的规则集合来决定什么样的字符应该产生什么样的单词。这个阶段处理杂乱的字符级别的细节，比如变量命名规则、注释和空格，它为下一阶段的处理准备好清楚的单词序列。

- *语法分析*

  读入一个单词序列并根据正在分析的语言语法判断它们是否代表一个有效的程序。如果程序有效，那么语法解析器会生成一些关于程序结构的附加信息（如一个解析树）。

4.3.1　词法分析

词法分析阶段通常相当直接。这可以通过正则表达式实现（因而也就是通过一台 NFA 实现），因为它把字符序列与一些规则简单匹配以判断那些字符是否为关键字、变量名、运算符或者其他什么符号。下面是一些快速但是不整洁的 Ruby 代码，可以把一个 Simple 程序断成单词：

```ruby
class LexicalAnalyzer < Struct.new(:string)
  GRAMMAR = [
    { token: 'i', pattern: /if/         }, # if 关键字
    { token: 'e', pattern: /else/       }, # else 关键字
    { token: 'w', pattern: /while/      }, # while 关键字
    { token: 'd', pattern: /do-nothing/ }, # do-nothing 关键字
    { token: '(', pattern: /\(/         }, # 左小括号
    { token: ')', pattern: /\)/         }, # 右小括号
    { token: '{', pattern: /\{/         }, # 左大括号
    { token: '}', pattern: /\}/         }, # 右大括号
    { token: ';', pattern: /;/          }, # 分号
    { token: '=', pattern: /=/          }, # 等号
    { token: '+', pattern: /\+/         }, # 加号
    { token: '*', pattern: /\*/         }, # 乘号
    { token: '<', pattern: /</          }, # 小于号
    { token: 'n', pattern: /[0-9]+/     }, # 数字
    { token: 'b', pattern: /true|false/ }, # 布尔值
    { token: 'v', pattern: /[a-z]+/     }  # 变量名
  ]

  def analyze
    [].tap do |tokens|
      while more_tokens?
        tokens.push(next_token)
      end
    end
  end

  def more_tokens?
    !string.empty?
  end

  def next_token
    rule, match = rule_matching(string)
    self.string = string_after(match)
    rule[:token]
  end

  def rule_matching(string)
    matches = GRAMMAR.map { |rule| match_at_beginning(rule[:pattern], string) }
    rules_with_matches = GRAMMAR.zip(matches).reject { |rule, match| match.nil? }
    rule_with_longest_match(rules_with_matches)
  end

  def match_at_beginning(pattern, string)
    /\A#{pattern}/.match(string)
  end

  def rule_with_longest_match(rules_with_matches)
    rules_with_matches.max_by { |rule, match| match.to_s.length }
  end

  def string_after(match)
    match.post_match.lstrip
  end
end
```



这个实现使用单个字符作为单词——w 的意思是“while 关键字”，+ 的意思是“加号”，以此类推——因为我们准备把这些单词提供给 PDA，而我们 Ruby 模拟的 PDA 期望以字符作为输入。

单字符的单词足以应付基本的演示需要了，这里我们不需要保留变量名或者字面值。但是在真正的解析器里，我们就需要合适的数据结构表示单词，这样它们才能在传达“某个不知名的变量”或“某个未知的布尔值”之外包含更多的信息。

通过使用 Simple 代码组成的字符串创建 LexicalAnalyzer 实例，然后调用它的 #analyze 方法，我们可以获得一个由单词组成的数组，这个数组说明了如何把代码断成关键字、运算符、标点以及其他语法：

```
>> LexicalAnalyzer.new('y = x * 7').analyze
=> ["v", "=", "v", "*", "n"]
>> LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze
=> ["w", "(", "v", "<", "n", ")", "{", "v", "=", "v", "*", "n", "}"]
>> LexicalAnalyzer.new('if (x < 10) { y = true; x = 0 } else { do-nothing }').analyze
=> ["i", "(", "v", "<", "n", ")", "{", "v", "=", "b", ";", "v", "=", "n", "}", "e",
"{", "d", "}"]
```



词法分析要按照最长匹配选择规则进行，否则会造成变量名被错误地识别为关键字：

```
>> LexicalAnalyzer.new('x = false').analyze
=> ["v", "=", "b"]
>> LexicalAnalyzer.new('x = falsehood').analyze
=> ["v", "=", "v"]
```

解决这个问题还有其他的方法。一种就是在规则中使用限制性更强的正则表达式：如果布尔值的规则使用模式 /(true|false)(?![a-z])/，那它就不会首先匹配字符串 'falsehood' 了。

4.3.2 语法分析

把字符串转成单词之后，难一些的问题就是确定这些单词是否表示一个语法有效的 Simple 程序了。我们不能使用正则表达式或者 NFA——Simple 的语法允许任意的括号嵌套，而我们已经知道有限自动机的能力不足以识别这样的语言。但是使用下推自动机是可以识别单词的有效序列的，所以下面来看看如何构造一台下推自动机。

首先，我们需要一个语法描述单词如何组合形成程序。下面是基于 2.6 节中 Treetop 语法结构的一部分 Simple 语法：

```
<语句>        ::= <while> | <赋值>
```

```
<while>        ::= 'w' '(' <表达式> ')' '{' <语句> '}'
<赋值>         ::= 'v' '=' <表达式>
<表达式>       ::= <小于表达式>
<小于表达式>   ::= <乘> '<' <小于表达式> | <乘>
<乘>           ::= <名词> '*' <乘> | <名词>
<名词>         ::= 'n' | 'v'
```

这叫作*上下文无关文法*（Context-Free Grammar，CFG）。[7] 每一条规则的左边是一个符号，右边是一个或多个符号序列和单词。例如，规则 <语句> `::= <while> | <赋值>` 的意思是一个 Simple 语句要么是 `while` 循环要么是一个赋值，而 <赋值> `::= 'v' '=' <表达式>` 的意思是一个赋值语句由一个变量名后面跟上一个等号和一个表达式组成。

CFG 是一个 Simple 结构的静态描述，但我们把它看成一个*生成* Simple 程序的规则集合。从"<语句>"开始，我们应用文法规则递归展开符号直到只剩下单词为止。下面是根据规则完全展开"<语句>"的方式之一：

```
<语句> → <赋值>
       → 'v' '=' <表达式>
       → 'v' '=' <小于表达式>
       → 'v' '=' <乘>
       → 'v' '=' <名词> '*' <乘>
       → 'v' '=' 'v' '*' <乘>
       → 'v' '=' 'v' '*' <名词>
       → 'v' '=' 'v' '*' 'n'
```

这表明 `'v' '=' 'v' '*' 'n'` 在语法上有效，但我们要的是相反方向的能力：能*识别*有效的程序，而不是生成它们。在由词法分析得到一串单词的时候，我们想要知道是否可以按照某种顺序应用文法规则把"<语句>"扩展成这些单词。幸好，有办法把上下文无关文法转换成能做出这种判断的非确定性下推自动机。

把一个 CFG 转换成 PDA 的方法如下。

(1) 选取一个字符表示文法中的每个符号。在这种情况下，我们使用每个符号的大写首字母——`S` 表示"<语句>"，`W` 表示 `<while>`，以此类推——这是为了与我们已经用来作为单词的小写字符区分开。

(2) 使用 PDA 的栈存储表示文法符号的字符（`S`、`W`、`A`、`E`……）和单词（`w`、`v`、`=`、`*`、……）。PDA 启动的时候，立即把一个符号推入栈中，这个符号表示它正在试图识别的结构。我们想要识别 Simple 语句，所以 PDA 开始时要把 `S` 推入栈中：

```
>> start_rule = PDARule.new(1, nil, 2, '$', ['S', '$'])
=> #<struct PDARule ...>
```

注 7：文法是"上下文无关的"指它的规则没有提到文法可能出现的上下文；一个赋值语句不管周围是什么单词，它总是包含一个变量名、赋值符号和表达式。不是所有的语言都可以用这种文法描述，但几乎所有的编程语言都可以。

(3) 把文法规则转换成无需任何输入就能扩展栈顶符号的 PDA 规则。每一个文法规则描述了如何把一个符号扩展成由其他符号和单词组成的序列，而我们可以把这个描述转换成一个 PDA 规则，它把一个代表特定符号的字符弹出栈并把其他字符推入栈中：

```
>> symbol_rules = [
     # <statement> ::= <while> | <assign>
     PDARule.new(2, nil, 2, 'S', ['W']),
     PDARule.new(2, nil, 2, 'S', ['A']),

     # <while> ::= 'w' '(' <expression> ')' '{' <statement> '}'
     PDARule.new(2, nil, 2, 'W', ['w', '(', 'E', ')', '{', 'S', '}']),

     # <assign> ::= 'v' '=' <expression>
     PDARule.new(2, nil, 2, 'A', ['v', '=', 'E']),

     # <expression> ::= <less-than>
     PDARule.new(2, nil, 2, 'E', ['L']),

     # <less-than> ::= <multiply> '<' <less-than> | <multiply>
     PDARule.new(2, nil, 2, 'L', ['M', '<', 'L']),
     PDARule.new(2, nil, 2, 'L', ['M']),

     # <multiply> ::= <term> '*' <multiply> | <term>
     PDARule.new(2, nil, 2, 'M', ['T', '*', 'M']),
     PDARule.new(2, nil, 2, 'M', ['T']),

     # <term> ::= 'n' | 'v'
     PDARule.new(2, nil, 2, 'T', ['n']),
     PDARule.new(2, nil, 2, 'T', ['v'])
   ]
=> [#<struct PDARule ...>, #<struct PDARule ...>, ...]
```

例如，赋值语句的规则说的是“< 赋值 >”符号可以扩展成单词 v、= 以及后面的“< 表达式 >”符号，因此我们有一个对应的 PDA 规则，它可以自发地从栈中弹出 A 并推入字符 v=E。“< 语句 >”规则说的是我们可以把“< 语句 >”符号用一个 <while> 或者“< 赋值 >”替换；我们已经把它转换成了一个 PDA 规则，它把一个 S 从栈中弹出，然后用一个 W 替换，而另一条规则是把 S 从弹出然后推入 A。

(4) 为每一个单词符号赋予一个 PDA 规则，这个规则从输入读取字符然后把它从栈中弹出：

```
>> token_rules = LexicalAnalyzer::GRAMMAR.map do |rule|
     PDARule.new(2, rule[:token], 2, rule[:token], [])
   end
=> [#<struct PDARule ...>, #<struct PDARule ...>, ...]
```

这些单词规则与符号规则的工作方式相反。符号规则试图让栈变大，有时候会推入一些字符以替换已经弹出的；单词规则总是让栈更小，随着栈的变小处理输入。

(5) 最后，生成一个 PDA 规则，在栈变成空时它允许机器进入接收状态：

```
>> stop_rule = PDARule.new(2, nil, 3, '$', ['$'])
=> #<struct PDARule ...>
```

现在我们可以使用这些规则构建一台 PDA，输入一个由单词组成的字符串看它是否能够识别。由 Simple 语法生成的规则是非确定性的（每当字符 S、L、M 或者 T 处于栈顶的时候，就会有多个可用的规则），因此它只能是一台 NPDA。

```
>> rulebook = NPDARulebook.new([start_rule, stop_rule] + symbol_rules + token_rules)
=> #<struct NPDARulebook rules=[...]>
>> npda_design = NPDADesign.new(1, '$', [3], rulebook)
=> #<struct NPDADesign ...>
>> token_string = LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze.join
=> "w(v<n){v=v*n}"
>> npda_design.accepts?(token_string)
=> true
>> npda_design.accepts?(LexicalAnalyzer.new('while (x < 5 x = x * }').analyze.join)
=> false
```

为了准确地表示整个过程，下面是向这台 NPDA 输入字符串 `'w(v<n){v=v*n}'` 后的一个可能执行：

状态	是否接受	栈的内容	剩余输入	动作
1	否	$	w(v<n){v=v*n}	推入 S，转移到状态 2
2	否	S$	w(v<n){v=v*n}	弹出 S，推入 W
2	否	W$	w(v<n){v=v*n}	弹出 W，推入 w(E){S}
2	否	w(E){S}$	w(v<n){v=v*n}	读取 w，弹出 w
2	否	(E){S}$	(v<n){v=v*n}	读取 (，弹出 (
2	否	E){S}$	v<n){v=v*n}	弹出 E，推入 L
2	否	L){S}$	v<n){v=v*n}	弹出 L，推入 M<L
2	否	M<L){S}$	v<n){v=v*n}	弹出 M，推入 T
2	否	T<L){S}$	v<n){v=v*n}	弹出 T，推入 v
2	否	v<L){S}$	v<n){v=v*n}	读取 v，弹出 v
2	否	<L){S}$	<n){v=v*n}	读取 <，弹出 <
2	否	L){S}$	n){v=v*n}	弹出 L，推入 M
2	否	M){S}$	n){v=v*n}	弹出 M，推入 T
2	否	T){S}$	n){v=v*n}	弹出 T，推入 n
2	否	n){S}$	n){v=v*n}	读取 n，弹出 n
2	否	){S}$	){v=v*n}	读取)，弹出)
2	否	{S}$	{v=v*n}	读取 {，弹出 {
2	否	S}$	v=v*n}	弹出 S，推入 A
2	否	A}$	v=v*n}	弹出 A，推入 v=E
2	否	v=E}$	v=v*n}	读取 v，弹出 v
2	否	=E}$	=v*n}	读取 =，弹出 =
2	否	E}$	v*n}	弹出 E，推入 L

（续）

状态	是否接受	栈的内容	剩余输入	动作
2	否	L}$	v*n}	弹出 L，推入 M
2	否	M}$	v*n}	弹出 M，推入 T*M
2	否	T*M}$	v*n}	弹出 T，推入 V
2	否	v*M}$	v*n}	读取 v，弹出 v
2	否	*M}$	*n}	读取 *，弹出 *
2	否	M}$	n}	弹出 M，推入 T
2	否	T}$	n}	弹出 T，推入 n
2	否	n}$	n}	读取 n，弹出 n
2	否	}$	}	读取 }，弹出 }
2	否	$		转移到 3
3	是	$		—

这个执行过程的跟踪向我们展示了机器在符号和单词规则之间的摇摆：符号规则不断地扩展栈顶符号，直到此符号被一个单词取代，然后单词规则再对栈（和输入）进行处理，直到遇到一个符号为止。只要输入字符串能够由文法规则生成，这样的反复就能得到一个空栈。[8]

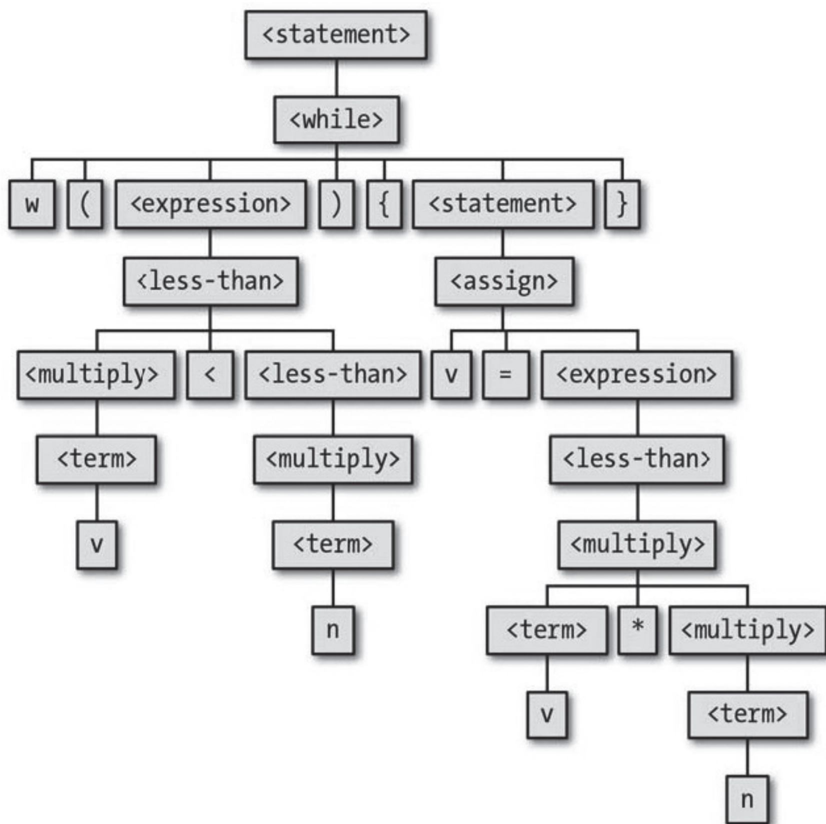
在每一步执行中 PDA 是怎么知道选择哪个规则的呢？这是非确定性的力量：我们模拟的 NPDA 对所有可能的规则进行尝试，因此只要存在*某种*方式能得到空栈，我们就能找到它。

4.3.3　实践性

这个分析的过程依赖于非确定性，但在实际程序中，最好能避免非确定性，因为一个确定性的 PDA 模拟起来要比非确定性的快得多而且容易得多。幸运的是，在每个阶段几乎都可以使用输入单词本身决定该应用哪个符号规则，这样就可能把非确定性去掉——这个技术叫作*递推*（lookahead）——但这让从 CFG 到 PDA 的转换更为复杂。

只能*识别*有效程序也不够好。就像我们在 2.6 节看到的那样，解析一个程序的要领就是把程序转成一个能用来做一些有用事情的结构化表示。在实践中，我们可以让 PDA 模拟记录它到达接受状态过程中的规则序列，以此来创建结构化表示，这个规则序列提供了构建一个分析树所需的足够信息。例如，上面的执行序列展示了为了形成需要的单词序列如何展开栈顶的符号，并且告诉了我们字符串 'w(v<n){v=v*n}' 的解析树形状：

注 8：这个算法叫作 LL 分析。第一个 L 代表“从左到右”，因为输入字符串是按这个方向读取的；第二个 L 代表“左侧优先推导”，因为总是栈中最左边的（也就是最上面的）符号得到扩展。



4.4 有多少能力

在这一章中，我们见到了两个新的计算能力的级别：DPDA 比 DFA 和 NFA 更强大，而 NPDA 要比 DPDA 更强大。能访问栈之后，看起来下推自动机比有限自动机要强大和复杂一些。

拥有栈的主要结果就是能识别某些有限自动机不能识别的语言了，如回文和平衡括号字符串。栈提供的无限存储使 PDA 能在计算中记住任意数量的信息并在随后再次使用它。

与有限自动机不同，PDA 可以在没有任何输入的情况下无限循环，这虽然不是很有用，但是比较少见。DFA 只能通过处理输入字符来改变状态，而 NFA 尽管可以自发地通过自由移动改变状态，但它只能在回到起点之前进行有限次数的自由移动。另一方面，PDA 可以保持在一个状态并不断地把字符推入栈中，永远也不会重复同样的配置。

在某种程度上，下推自动机还能控制自己。在规则和栈之间有一个反馈环——栈的内容影响机器应该遵守的规则，而按照某个规则执行也会影响栈的内容——这允许 PDA 在栈中

存储一些信息，这些信息可以影响它将来的执行。有限自动机依赖于类似的规则和当前状态之间的反馈，但这个反馈作用要小一些，因为当前状态在改变之后就完全被遗忘了，而把字符推入栈中可以把老的内容保存起来供以后使用。

因此 PDA 确实是要强大一些，但它的限制是什么呢？即使我们只对看到的模式匹配应用感兴趣，下推自动机仍然严重受限于栈的工作方式。在栈顶字符之下的内容没有办法随机访问，因此如果机器想要读取埋在栈中间的一个字符，就得弹出这个字符上面所有的东西。一旦字符被弹出，就永远消失了。我们设计了一台 PDA 以识别由等量的 `a` 和 `b` 组成的字符串，但没法修改它以识别由等量的三种字符组成的字符串（`'abc'`、`'aabbcc'`、`'aaabbbccc'`……），因为关于 `a` 的数量的信息在对 `b` 计数的过程中被破坏了。

撇开能用的向栈中推送字符的次数，栈的后进先出属性也会引起信息存储和获取的问题。PDA 能识别回文，但它不能识别 `'abab'` 和 `'baaabaaa'` 这样“双倍”的字符串，因为一旦信息被推入到栈中，就只能以相反的顺序处理了。

如果我们抛开识别字符串的特定问题，而把这些机器看成通用目的的计算机，就可以看到 DFA、NFA 和 PDA 还远远不够有用。首先，它们都没有像样的输出机制：它们通过进入接受状态表达成功，但不能输出哪怕一个字符（更不用说由字符组成的字符串了）来表示详细的结果。无法将信息发送回世界意味着它们连把两个数相加这样的简单算法都实现不了。而像有限自动机一样，PDA 有一个固定的程序；没有明显的方法构建出一台 PDA 能以某种方式从输入读取一个程序然后运行。

所有这些缺点意味着我们需要一个更好的计算模型，去真正地研究计算机能干什么，而这正是下一章的内容。

第5章 终极机器

在第 3 章和第 4 章，我们研究了简单计算模型的能力。我们已经看到如何识别复杂性逐渐增加的字符串、如何匹配正则表达式，以及如何解析编程语言，而且都是使用不太复杂的基本机器完成的。

但我们也看到，这些机器——有限自动机和下推自动机——都有很严格的限制，这些限制影响了它们作为现实计算模型的使用。我们的小型系统还要多强大，才能摆脱这些限制并完成正常计算机的所有工作呢？它还要多复杂才能对 RAM 或硬盘的行为以及合适的输出机制建模呢？怎么才能设计一台能实际运行程序而不总是执行某个硬编码任务的机器呢？

20 世纪 30 年代，阿兰·图灵（Alan Turing）致力于从本质上解决这个问题。在那个年代，单词 computer 意味着一个人，通常是一个女人，她手工重复着一系列繁重的数学性操作以执行长长的计算。图灵当时正在寻找一种理解和描述“人肉计算机”操作特征的方法，这样同样的工作就可以完全由机器执行。本章，我们将看到图灵关于设计最简单的“自动化机器”的思想，这一机器具有手工计算的全部能力和复杂性。

5.1 确定型图灵机

在第 4 章，我们通过给一台有限自动机赋予一个作为外部存储的栈，增强了它的计算能力。与由机器状态提供的有限内部存储相比，栈的真正优点是能动态增长以适应任意数量的信息，从而使下推自动机能够处理那些需要存储任意数量数据的问题。

但是，外部存储这种特殊的形式给如何使用存储之后的数据带来了限制。通过把栈替换成更灵活的存储机制，我们可以消除这些限制并进一步提高能力。

5.1.1 存储

> 计算通常可以通过在纸上写某些符号完成。我们可以把这张纸想象成小朋友的算术本，它被划分成了一个个方格。在初等算术里，我们有时也会使用纸的二维特性。但这种使用通常是可以避免的，并且我认为纸的二维性不是计算的本质，而且相信大家也赞同我这一观点。我假定计算是在一张一维的纸上完成的，比如在一条分成方格的纸带上完成。
>
> ——阿兰·图灵，《论可计算数及其在判定性问题上的应用》，
> http://dx.doi.org/10.1112/plms/s2-42.1.23o

图灵的做法是给一台机器配上一条无限长的空纸带（实际上是一个两端都能随需增长的一维数组），并且允许在纸带上的任意位置读写字符。一条纸带既做存储又做输入：可以在纸带上预先填满字符串当作输入，然后机器在执行过程中可以读取这些字符并在必要的时候覆盖它们。

能访问一条无限长纸带的有限状态自动机叫作*图灵机*（Turing Machine，TM ）。这个名字通常指一条拥有确定性规则的机器，但我们也可以毫无歧义地叫它*确定型图灵机*（Deterministic Turing Machine，DTM）。

我们已经知道，下推自动机只能访问其外部存储的一个固定位置（栈的顶部），但这似乎对图灵机来说限制性太强了。提供一条纸带的目的就是允许在纸带上的任何位置存储任意量的数据，并以任意顺序读取，那么我们如何设计一台能与整条纸带交互的机器呢？

一种选择是让纸带可以被随机寻址访问，就像计算机的 RAM 一样给每个方格标记一个独立的数字地址，这样机器可以立即读取和写入任何位置。这增加了不必要的复杂性，而且需要规划出细节上的东西，比如如何给一条无限纸带的所有方格分配地址，以及在它需要访问方格时如何指定方格的地址。

传统的图灵机不是这样，而是使用更简单的安排：用一个*纸带头*（tape head）指向纸带的一个特定位置，并且只能在那个位置读取或写入字符。每一步计算之后，纸带头都可以向左或者向右移动一个方格，这意味着一台图灵机为了到达远处的位置只能费力地在纸带上往复移动。使用移动缓慢的纸带头不会影响机器访问纸带上任何数据的能力，只会影响花费的时间，因此为了保持简单付出这个代价是值得的。

能访问纸带之后，除了能够接受或者拒绝字符串，我们又能解决新的问题了。例如，我们可以设计一台在纸带上就地递增一个二进制数的 DTM。为此，我们需要知道如何递增一个二进制数的一位数字。幸好这很简单：如果这位的数字是 0，就用 1 替换；如果这位数是 1，就用 0 替换，然后立即使用同样的方法增加它左边的数字（“进 1 位”）。图灵机只需要使用这个过程递增二进制数的最右位，然后把纸带头移到起始位置。

- 给机器赋予三个状态（状态 1、状态 2、状态 3），状态 3 作为接受状态。

- 机器从状态 1 开始，纸带头指向一个二进制数的最右位。
- 处于状态 1 并且读到一个 0（或者空白）时，就用 1 覆写，把纸带头向右移，然后回到状态 2。
- 处于状态 1 并且读到一个 1 时，就用 0 覆写，然后把纸带头向左移。
- 处于状态 2 并且读到一个 0 或者 1 时，就把纸带头向右移。
- 处于状态 2 并且读到空白时，就把纸带头向左移并转移到状态 3。

在机器试图递增一位数字的时候，它处于状态 1，在移回起始位置时处于状态 2，结束的时候处于状态 3。下面是初始纸带上字符串为 `'1011'` 时对机器执行的跟踪。纸带头当前指向的字符会由括号包围，而下划线表示输入字符串某一端的空白方格。

状态	是否接受	纸带内容	动作
1	否	`_101(1)__`	写入 0，左移
1	否	`__10(1)0_`	写入 0，左移
1	否	`___1(0)00`	写入 1，右移，转移到 2
2	否	`__11(0)0_`	右移
2	否	`_110(0)__`	右移
2	否	`1100(_)__`	左移，转移到状态 3
3	是	`_110(0)__`	—



严格来说，把纸带头移回它的初始位置并不必要（如果我们把状态 2 作为接受状态，则一旦机器成功地把 0 替换成 1，它会立即停止，而纸带仍会包含正确的结果），但这是一个值得要的特性，因为它把纸带头放到位之后，机器只要简单地把状态改变回状态 1 就可以再次运行。通过多次运行机器，我们可以不断递增存储在纸带上的数。这个功能可以重用，作为更大机器的一部分，比如说把两个二进制数相加或相乘。

5.1.2 规则

> 让我们想象一下，由机器执行的操作被分解成“简单的操作”，这些操作都非常基本，以至于无法想象它们能进一步分解。……操作实际上是由计算者的思维状态和被观察的符号决定的……具体来讲，操作执行之后，计算者的思维状态就确定了。
>
> 我们现在可以构造一台做这种计算者工作的机器了。
>
> ——阿兰·图灵，《论可计算数及其在判定性问题上的应用》

在每一步计算中，可能都有几个“简单的操作”需要图灵机执行：在纸带头的当前位置读取字符，在那个位置写入一个新字符，把纸带头左移或者右移，或者改变状态。简单起

见，我们没有为所有这些动作指定不同种类的规则，而只是像处理下推自动机时那样，只设计了一种能灵活适应各种条件的规则格式。

这个统一的规则格式有 5 部分：

- 机器的当前状态；
- 必须出现在纸带头当前位置的字符；
- 机器的下一状态；
- 要写入纸带头当前位置的字符；
- 写入纸带之后纸带头的移动方向（向左还是向右）。

这里我们假设一台图灵机每次执行规则，都要改变状态并向纸带写一个字符。就像通常对状态机的处理那样，如果我们想要一个规则不实际改变状态，可以让“下一个状态”与当前状态相同；与之类似的是，如果想要一个规则不改变纸带内容，可以把与读到的字符一样的字符写入纸带。

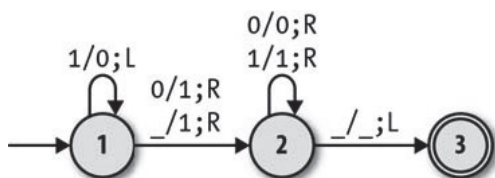


我们还假设了纸带头每步都要移动。这就不太可能书写一个不移动纸带头就更新状态或者纸带内容的规则，但我们可以通过一个规则做出需要的改变以得到同样的效果，然后再通过一个规则把纸带头移回原始位置。

递增一个二进制数的图灵机写成这种类型的话将有 6 个规则：

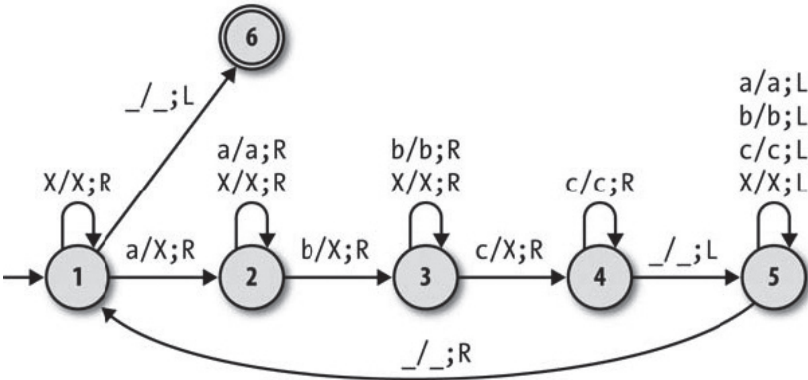
- 处于状态 1 并且读入一个 0 时，写入一个 1，右移，然后进入状态 2；
- 处于状态 1 并且读入一个 1 时，写入一个 0，左移，然后保持在状态 1；
- 处于状态 1 并且读到一个空白时，写入一个 1，右移，然后进入状态 2；
- 处于状态 2 并且读到一个 0 时，写入一个 0，右移，然后保持在状态 2；
- 处于状态 2 并且读入一个 1 时，写入一个 1，右移，然后保持在状态 2；
- 处于状态 2 并且读到一个空白时，写入一个空白，左移，然后进入状态 3。

与在有限自动机和下推自动机中使用的图类似，我们也可以展示机器的状态和规则：



事实上，除去箭头上的标签，这很像一个 DFA 示意图。标签 `a/b;L` 表示一条规则，它从纸带上读取字符 `a`，写入字符 `b`，然后把纸带头向左移动一个方格；标签 `a/b;R` 表示的规则几乎一样，只是会把纸带头向右而不是向左移动。

我们看一下如何使用图灵机解决下推自动机无法处理的字符串识别问题：要识别的字符串包含一个或者多个字符 a，后面跟随着同样数目的 b 和 c（如 'aaabbbccc'）。解决这个问题的图灵机有 6 个状态和 16 个规则：



它大致像这样工作：

(1) 通过不断把纸带头向右移扫描输入字符串，直到发现一个 a 为止，然后通过用 X 替换 a 来把它删除（状态 1）；
(2) 向右扫描寻找一个 b，然后删除（状态 2）；
(3) 向右扫描寻找一个 c，然后删除（状态 3）；
(4) 向右扫描寻找输入字符串的结尾（状态 4），然后向左扫描寻找输入字符串的开头（状态 5）；
(5) 重复这些步骤，直到所有的字符都已被删除为止。

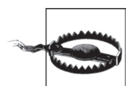
如果输入字符串是由一个或多个字符 a 以及同样数目的 b 和 c 组成的，那么机器将会重复跨越整个字符串几次，每次跨越都会删除一个字符，然后在整个字符串都被删掉的时候进入到一个接受状态。下面是在输入为 'aabbcc' 时的执行跟踪。

状态	是否接受	纸带内容	动作
1	否	______(a)abbcc_	写入 X，右移，转移到状态 2
2	否	_____X(a)bbcc__	右移
2	否	____Xa(b)bcc___	写入 X，右移，转移到状态 3
3	否	___XaX(b)cc____	右移
3	否	__XaXb(c)c_____	写入 X，右移，转移到状态 4
4	否	_XaXbX(c)______	右移
4	否	XaXbXc(_)______	左移，转移到状态 5
5	否	_XaXbX(c)______	左移

（续）

状态	是否接受	纸带内容	动作
5	否	__XaXb(X)c_____	左移
5	否	___XaX(b)Xc____	左移
5	否	____Xa(X)bXc___	左移
5	否	_____X(a)XbXc__	左移
5	否	______(X)aXbXc_	左移
5	否	______(_)XaXbXc	右移，转移到状态 1
1	否	______(X)aXbXc_	右移
1	否	_____X(a)XbXc__	写入 X，右移，转移到状态 2
2	否	____XX(X)bXc___	右移
2	否	___XXX(b)Xc____	写入 X，右移，转移到状态 3
3	否	__XXXX(X)c_____	右移
3	否	_XXXXX(c)______	写入 X，右移，转移到状态 4
4	否	XXXXXX(_)______	左移，转移到状态 5
5	否	_XXXXX(X)______	左移
5	否	__XXXX(X)X_____	左移
5	否	___XXX(X)XX____	左移
5	否	____XX(X)XXX___	左移
5	否	_____X(X)XXXX__	左移
5	否	______(X)XXXXX_	左移
5	否	______(_)XXXXXX	右移，转移到状态 1
1	否	______(X)XXXXX_	右移
1	否	_____X(X)XXXX__	右移
1	否	____XX(X)XXX___	右移
1	否	___XXX(X)XX____	右移
1	否	__XXXX(X)X_____	右移
1	否	_XXXXX(X)______	右移
1	否	XXXXXX(_)______	左移，转移到状态 6
6	是	_XXXXX(X)______	—

这台机器能够工作是因为扫描阶段规则的准确选择。例如，机器处于状态 3（向右扫描并查找 c）的时候，它能执行的规则只能是移动纸带头经过 b 和 X。如果机器遇到了其他字符（如非期望的 a），它是没有规则可以遵守的，在这种情况下它会进入隐含的卡死状态停止执行，并会因此拒绝这个输入。



我们通过假设输入只包含字符 a、b 和 c 来保持简单，但如果不是这样，机器也不会正常工作。例如，它会接受字符串 'XaXXbXXXc'，即使这个字符串本来应该被拒绝。为了正确地处理这种输入，我们需要增加更多的规则和状态扫描整个字符串，以检查在机器开始删除字符之前这个字符串不包含任何非期望的字符。

5.1.3 确定性

对于设计成确定性的一台特定的图灵机，它只能遵守和确定性下推自动机一样的约束（参见 4.1.3 节），但这次我们不用担心自由移动，因为图灵机没有自由移动。

要根据图灵机的当前状态和当前纸带头下的字符来选择它的下一个动作，因此一台确定性机器只能有由状态和字符组合成的一个规则——“无矛盾”规则，这是为了避免下一个动作有任何的歧义。简单起见，我们会像处理 DPDA 时那样，放松“无遗漏”规则，并假设在没有规则可用的时候机器可以进入一个隐含的卡死状态，而不是坚持对于每一个可能的情况都要有一个规则。

5.1.4 模拟

我们已经对一台确定性图灵机应该如何工作有了很好的认识，现在来构建一个 Ruby 的模拟以便可以看到它的执行。

第一步是实现图灵机的纸带。很显然这个实现不得不存储写到纸带上的字符，但它还需要记住纸带头的当前位置，以便模拟出来的机器可以读取当前字符，在当前位置写入一个新的字符，并左右移动纸带头到达其他位置。

做到这一点的一个优雅方式是把纸带分成三部分（纸带头左边的全部字符、纸带头下的一个字符、右边的所有字符），每一部分分别存储。这让读写当前字符变得非常容易，而移动纸带头可以通过在所有三个部分之间慢慢移动字符实现。例如，向右移动一个方格，意味着之前的当前字符成为了纸带头左边的最后一个字符，而之前纸带头右边的第一个字符成为了当前字符。

我们的实现还必须维护纸带无限长而且填满空白方格的假象，但幸好并不需要一个无限大的数据结构。在任意给定时刻唯一能被读取的是纸带头下的位置，因此在纸带头移动超出了已经写在纸带上的有限数目的非空字符时。我们只需安排一个空白字符出现。为此，我们需要预先知道哪个字符代表一个空白方格，然后只要进入到纸带上未经探索的区域，就可以让这个字符自动出现在纸带头下。

因此，一条纸带的基本表示看起来像这样：

```
class Tape < Struct.new(:left, :middle, :right, :blank)
  def inspect
    "#<Tape #{left.join}(#{middle})#{right.join}>"
  end
end
```

因此可以创建一条纸带并读取纸带头下面的字符：

```
>> tape = Tape.new(['1', '0', '1'], '1', [], '_')
=> #<Tape 101(1)>
>> tape.middle
=> "1"
```

我们可以增加操作，向当前纸带位置[1]写入并把纸带头左右移动：

```
class Tape
  def write(character)
    Tape.new(left, character, right, blank)
  end

  def move_head_left
    Tape.new(left[0..-2], left.last || blank, [middle] + right, blank)
  end

  def move_head_right
    Tape.new(left + [middle], right.first || blank, right.drop(1), blank)
  end
end
```

现在可以向纸带写入，并来回移动纸带头：

```
>> tape
=> #<Tape 101(1)>
>> tape.move_head_left
=> #<Tape 10(1)1>
>> tape.write('0')
=> #<Tape 101(0)>
>> tape.move_head_right
=> #<Tape 1011(_)>
>> tape.move_head_right.write('0')
=> #<Tape 1011(0)>
```

在第 4 章，我们使用*配置*一词来代表下推自动机状态和栈的组合，同样的理念在这里也会很有帮助。可以说一个*图灵机的配置*是一个状态和一条纸带的组合，并且可以直接处理这些配置的图灵机规则：

```
class TMConfiguration < Struct.new(:state, :tape)
end

class TMRule < Struct.new(:state, :character, :next_state,
                          :write_character, :direction)
  def applies_to?(configuration)
    state == configuration.state && character == configuration.tape.middle
  end
end
```

只有在机器的当前状态和纸带头下的当前字符与其表达式匹配时，规则才能应用：

注 1：就像栈一样，纸带是纯功能性的：写入纸带和移动纸带头都是非破坏性操作，只会返回一个新的 Tape，而不是更新已有的对象。

```
>> rule = TMRule.new(1, '0', 2, '1', :right)
=> #<struct TMRule
     state=1,
     character="0",
     next_state=2,
     write_character="1",
     direction=:right
   >
>> rule.applies_to?(TMConfiguration.new(1, Tape.new([], '0', [], '_')))
=> true
>> rule.applies_to?(TMConfiguration.new(1, Tape.new([], '1', [], '_')))
=> false
>> rule.applies_to?(TMConfiguration.new(2, Tape.new([], '0', [], '_')))
=> false
```

知道一个规则能在一个特定的配置下应用之后，我们需要能够通过写入一个新字符、移动纸带头以及按照规则改变机器状态来更新该配置：

```
class TMRule
  def follow(configuration)
    TMConfiguration.new(next_state, next_tape(configuration))
  end

  def next_tape(configuration)
    written_tape = configuration.tape.write(write_character)

    case direction
    when :left
      written_tape.move_head_left
    when :right
      written_tape.move_head_right
    end
  end
end
```

这些代码看起来工作得很好：

```
>> rule.follow(TMConfiguration.new(1, Tape.new([], '0', [], '_')))
=> #<struct TMConfiguration state=2, tape=#<Tape 1(_)>>
```

DTMRulebook 的实现几乎与 DFARulebook 和 DPDARulebook 一样，只是方法 #next_configuration 没有用字符作为参数，这是因为没有外部的输入可供读取字符（只有纸带，而纸带已经是配置的一部分了）：

```
class DTMRulebook < Struct.new(:rules)
  def next_configuration(configuration)
    rule_for(configuration).follow(configuration)
  end

  def rule_for(configuration)
    rules.detect { |rule| rule.applies_to?(configuration) }
  end
end
```

我们现在可以为“递增二进制数”的图灵机创建一个 DTMRulebook，并手工单步执行一些配置：

```
>> rulebook = DTMRulebook.new([
     TMRule.new(1, '0', 2, '1', :right),
     TMRule.new(1, '1', 1, '0', :left),
     TMRule.new(1, '_', 2, '1', :right),
     TMRule.new(2, '0', 2, '0', :right),
     TMRule.new(2, '1', 2, '1', :right),
     TMRule.new(2, '_', 3, '_', :left)
   ])
=> #<struct DTMRulebook rules=[...]>
>> configuration = TMConfiguration.new(1, tape)
=> #<struct TMConfiguration state=1, tape=#<Tape 101(1)>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=1, tape=#<Tape 10(1)0>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=1, tape=#<Tape 1(0)00>>
>> configuration = rulebook.next_configuration(configuration)
=> #<struct TMConfiguration state=2, tape=#<Tape 11(0)0>>
```

把所有这些封装成一个 DTM 类很方便，这样就像第 2 章里实现小步语义时那样，我们可以有 #step 和 #run 方法：

```
class DTM < Struct.new(:current_configuration, :accept_states, :rulebook)
  def accepting?
    accept_states.include?(current_configuration.state)
  end

  def step
    self.current_configuration = rulebook.next_configuration(current_configuration)
  end

  def run
    step until accepting?
  end
end
```

我们现在有了一台确定型图灵机的模拟，因此给它一些输入试验一下：

```
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM ...>
>> dtm.current_configuration
=> #<struct TMConfiguration state=1, tape=#<Tape 101(1)>>
>> dtm.accepting?
=> false
>> dtm.step; dtm.current_configuration
=> #<struct TMConfiguration state=1, tape=#<Tape 10(1)0>>
>> dtm.accepting?
=> false
>> dtm.run
=> nil
>> dtm.current_configuration
```

```
=> #<struct TMConfiguration state=3, tape=#<Tape 110(0)_>>
>> dtm.accepting?
=> true
```

就像对待 DPDA 模拟一样，为了能优雅地处理卡死状态的图灵机我们需要再多做一些工作：

```
>> tape = Tape.new(['1', '2', '1'], '1', [], '_')
=> #<Tape 121(1)>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM ...>
>> dtm.run
NoMethodError: undefined method 'follow' for nil:NilClass
```

这次我们不需要一个卡死状态的特殊表示了。与 PDA 不同，图灵机没有外部输入，因此可以通过看它的规则手册和当前配置判断其是否处于卡死状态：

```
class DTMRulebook
  def applies_to?(configuration)
    !rule_for(configuration).nil?
  end
end

class DTM
  def stuck?
    !accepting? && !rulebook.applies_to?(current_configuration)
  end

  def run
    step until accepting? || stuck?
  end
end
```

现在模拟会注意到它卡住了并且自动停止：

```
>> dtm = DTM.new(TMConfiguration.new(1, tape), [3], rulebook)
=> #<struct DTM ...>
>> dtm.run
=> nil
>> dtm.current_configuration
=> #<struct TMConfiguration state=1, tape=#<Tape 1(2)00>>
>> dtm.accepting?
=> false
>> dtm.stuck?
=> true
```

只是为了好玩，下面是我们之前看到的用来识别 'aaabbbccc' 这样的字符串的图灵机：

```
>> rulebook = DTMRulebook.new([
     # 状态 1：向右扫描，查找 a
     TMRule.new(1, 'X', 1, 'X', :right), # 跳过 X
     TMRule.new(1, 'a', 2, 'X', :right), # 删除 a，进入状态 2
```

```
  TMRule.new(1, '_', 6, '_', :left),  # 查找空格，进入状态 6（接受）

  # 状态 2：向右扫描，查找 b
  TMRule.new(2, 'a', 2, 'a', :right), # 跳过 a
  TMRule.new(2, 'X', 2, 'X', :right), # 跳过 X
  TMRule.new(2, 'b', 3, 'X', :right), # 删除 b，进入状态 3

  # 状态 3：向右扫描，查找 c
  TMRule.new(3, 'b', 3, 'b', :right), # 跳过 b
  TMRule.new(3, 'X', 3, 'X', :right), # 跳过 X
  TMRule.new(3, 'c', 4, 'X', :right), # 删除 c，进入状态 4

  # 状态 4：向右扫描，查找字符串结束标记
  TMRule.new(4, 'c', 4, 'c', :right), # 跳过 c
  TMRule.new(4, '_', 5, '_', :left),  # 查找空格，进入状态 5

  # 状态 5：向左扫描，查找字符串开始标记
  TMRule.new(5, 'a', 5, 'a', :left),  # 跳过 a
  TMRule.new(5, 'b', 5, 'b', :left),  # 跳过 b
  TMRule.new(5, 'c', 5, 'c', :left),  # 跳过 c
  TMRule.new(5, 'X', 5, 'X', :left),  # 跳过 X
  TMRule.new(5, '_', 1, '_', :right)  # 查找空格，进入状态 1
  ])
=> #<struct DTMRulebook rules=[...]>
>> tape = Tape.new([], 'a', ['a', 'a', 'b', 'b', 'b', 'c', 'c', 'c'], '_')
=> #<Tape (a)aabbbccc>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [6], rulebook)
=> #<struct DTM ...>
>> 10.times { dtm.step }; dtm.current_configuration
=> #<struct TMConfiguration state=5, tape=#<Tape XaaXbbXc(c)_>>
>> 25.times { dtm.step }; dtm.current_configuration
=> #<struct TMConfiguration state=5, tape=#<Tape _XXa(X)XbXXc_>>
>> dtm.run; dtm.current_configuration
=> #<struct TMConfiguration state=6, tape=#<Tape _XXXXXXXX(X)_>>
```

这个实现很容易构建，只要我们有了表示纸带和规则手册的数据结构，模拟一台图灵机并不难。当然，阿兰·图灵特意让它们保持简单以便容易构建和推导，并且我们将在之后（5.4 节）看到实现的简单性也是一个重要属性。

5.2 非确定型图灵机

在 3.4 节中，我们看到非确定性没有让有限自动机有什么不同，而 4.2.2 节表明一台非确定性的下推自动机比一台确定性的能多做一些事情，这留给我们一个明显的关于图灵机的问题：增加不确定性[2]会使一台图灵机更强大吗？

答案是不会：一台非确定型图灵机并不能比一台确定型图灵机多做任何事情。下推自动机是个例外，因为 DFA 和 DTM 都有足够的能力模拟其非确定性的对应机器。有限自动机的

注 2：对于一台图灵机，“不确定性”意味着每个状态和字符的组合会允许多于一个的规则，因此从一个起始配置开始会有多个可能的执行路径。

一个状态能用来表示许多状态的组合，而图灵机的一条纸带能用来存储许多纸带的内容，但一个下推自动机的栈无法同时表示多个可能的栈。

因此，就像有限自动机一样，一台确定型图灵机可以模拟一台非确定型图灵机。使用纸带存储由图灵机配置适当编码后组成的一个队列，每一个配置都包含一个可能的当前状态和所模拟机器的纸带，模拟就靠它运行。模拟开始的时候，纸带上只存有一个配置，它表示所模拟机器的初始配置。模拟计算的每一步执行都是先读取队列前面的配置，找到能用的每一个规则，并使用这个规则生成新的配置，再把配置写回纸带放到队尾。一旦对每一个规则都这样做了，最前面的配置会被擦除，然后会再次对队列中的下一个配置进行处理。这个机器模拟的步骤会一直重复，直到队列前面的配置表示机器已经到达接受状态为止。

这个技术允许确定型图灵机按照广度优先的顺序探索被模拟机器的所有可能配置。如果对于非确定型图灵机来说存在一条执行路径到达一个接受状态，模拟就会找到它，就算其他路径会导致无限循环也没有关系。实际上把这个模拟实现为一个规则手册要求大量的细节，因此我们不会在这里进行尝试，但能够用确定型图灵机模拟就意味着我们不能仅仅通过增加非确定性就让一台图灵机更强大。

5.3　最大能力

确定型图灵机代表了从有限计算机器到全能机器的临界点。实际上，通过升级图灵机规范以使其更强大的任何尝试都注定失败，因为它们本来就有能力模拟任何潜在的增强了。[3] 尽管增加某些特性会使图灵机更小巧或者更高效，但无法从根本上增强它们的能力。

我们之前已经看到了对于非确定性来说为什么这是对的。现在来看一下对传统图灵机的 4 个其他扩展——内部存储、子例程、多纸带以及多维纸带——并领会为什么它们中没有一个可以增强计算能力。尽管涉及的模拟技术很复杂，但到最后，它们都只不过是编程方面的问题。

5.3.1　内部存储

为图灵机设计规则手册非常让人沮丧，因为它们缺少随机的内部存储。例如，我们经常想要机器把纸带头移动到一个特定的位置，读取存在那儿的字符，然后移动到另一个不同的部分，再根据之前读到的字符执行某个动作。表面看来，这似乎不太可能，因为没有地方能让机器“记住”那个字符——当然它仍旧写在纸带上，并且只要我们喜欢，就可以把纸带头移动回到那里再次对其读取，但只要纸带头从那个方格移开了，我们就再也不能根据它的内容触发一个规则了。

注 3：严格来讲，只有我们实际知道如何实现的增强才算数。如果赋予一台图灵机魔力，让它能立即推理出传统图灵机无法回答的问题的答案，它确实会变得更强大，但实际上，这是无法做到的。

如果图灵机有一些临时性的内部存储（可以叫它“RAM”“寄存器”“本地变量”，等等）会更方便，其中可以保存纸带当前方格的字符，而且即使以后纸带头已经完全移动到了不同的部分，也能对其引用。实际上，如果一台图灵机有这个能力，我们就没必要限制它存储纸带上的字符：它可以存储任何相关的信息，比如机器执行计算的中间结果，从而把我们从来回移动纸带头向纸带写回碎片数据的繁琐工作中解放出来。这个额外的灵活性好像能让图灵机执行新类型的任务了。

就像非确定性一样，为图灵机增加额外的内部存储确实会让某些任务更容易执行，但它并不能让机器做任何它本来不能完成的工作。把中间结果存在机器内部而不是纸带上的念头很容易消除，因为即使让纸带头来回移动访问这些信息要花费些工夫，用纸带存储这种信息也能工作得很好。但我们不得不更加认真地看待这个记忆字符的点，因为如果纸带头移动到其他地方之后就不能利用之前纸带方格里的内容的话，一台图灵机的作用会非常有限。

幸好图灵机有非常完美的内部存储——它的当前状态。图灵机可用的状态数目没有上限，但对于任意的特定规则集合来说，这个数目一定是有限的并且要预先决定好，因为无法在计算过程中创建新的状态。如果必要，我们可以设计一台拥有 100 个、1000 个，甚至 10 亿个状态的机器，然后使用当前状态记住从一步到下一步任意数量的信息。

这意味着免不了要复制规则适应多个状态，因为这些状态除了“记住”的信息不同外都是相同的。一台机器不是只用一个状态表示“向右扫描查找一个空白方格”，而是可以为“向右扫描查找一个空白方格（记住我之前读取到了一个 a）”设置一个状态，再为“向右扫描查找一个空白方格（记住我之前读取到了一个 b）”设置另一个状态，所有可能的字符都以此类推——字符数目也是有限的，所以这样的复制总是有限的。

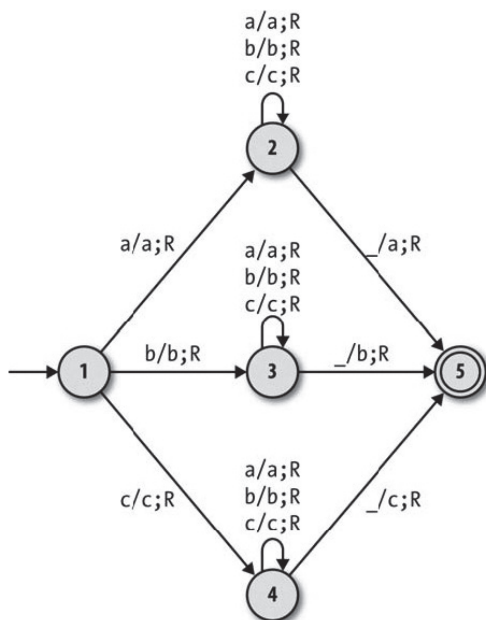
下面是一个使用这种技术的图灵机，它会把一个字符从字符串的开头复制到结尾：

```
>> rulebook = DTMRulebook.new([
     # 状态 1：从磁带读取第一个字符
     TMRule.new(1, 'a', 2, 'a', :right), # 记住 a
     TMRule.new(1, 'b', 3, 'b', :right), # 记住 b
     TMRule.new(1, 'c', 4, 'c', :right), # 记住 c

     # 状态 2：向右扫描，查找字符串结束标记（记住 a）
     TMRule.new(2, 'a', 2, 'a', :right), # 跳过 a
     TMRule.new(2, 'b', 2, 'b', :right), # 跳过 b
     TMRule.new(2, 'c', 2, 'c', :right), # 跳过 c
     TMRule.new(2, '_', 5, 'a', :right), # 找到空格，写 a
     # 状态 3：向右扫描，查找字符串结束标记（记住 b）
     TMRule.new(3, 'a', 3, 'a', :right), # 跳过 a
     TMRule.new(3, 'b', 3, 'b', :right), # 跳过 b
     TMRule.new(3, 'c', 3, 'c', :right), # 跳过 c
     TMRule.new(3, '_', 5, 'b', :right), # 找到空格，写 b

    # 状态 4：向右扫描，查找字符串结束标记（记住 c）
```

```
  TMRule.new(4, 'a', 4, 'a', :right), # 跳过 a
  TMRule.new(4, 'b', 4, 'b', :right), # 跳过 b
  TMRule.new(4, 'c', 4, 'c', :right), # 跳过 c
  TMRule.new(4, '_', 5, 'c', :right)  # 查找空格，写 c
 ])
=> #<struct DTMRulebook rules=[...]>
>> tape = Tape.new([], 'b', ['c', 'b', 'c', 'a'], '_')
=> #<Tape (b)cbca>
>> dtm = DTM.new(TMConfiguration.new(1, tape), [5], rulebook)
=> #<struct DTM ...>
>> dtm.run; dtm.current_configuration.tape
=> #<Tape bcbcab(_)>
```



除了它们每一个所表示的机器记住的字符串开头字符不同之外，这台机器的状态 2、3 和 4 几乎完全相同，并且在这种情况下，在到达末端的时候它们都做了一些不同的事情。



这台机器只对由字符 a、b、c 组成的字符串起作用。如果想要其对由字母表里任意字母组成的字符串起作用（或者字母数字字符，或者我们选择的更大集合），必须加入多得多的状态（为可能需要记住的每一个字符设置一个状态），还得加入多得多的与之匹配的规则。

如果用这种方式利用当前状态，我们可以设计出任凭纸带头来回移动仍能记住之前任何组合的图灵机，这实际上与给一台机器提供明确的“寄存器”作为内部存储有同样的能力，只不过代价是使用了大量的状态。

5.3.2 子例程

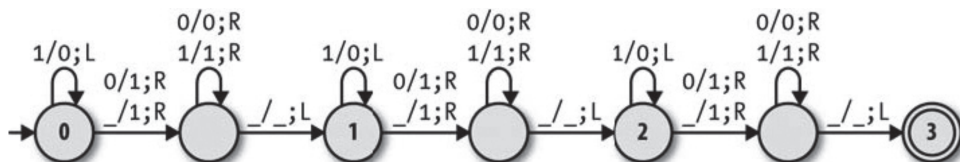
一台图灵机的规则手册是一个很长的、由极为低层次的指令组成的硬编码列表，因此在写这些规则时不忽略机器应该执行的高层次任务是很困难的。如果存在调用*子例程*的方法，设计一个规则手册会更容易一些：如果机器的某个部分能把所有这些规则存储成子例程，比如说叫"递增一个数"，那么我们的规则手册就不需要手工拼凑这些指令，而只需要说"现在递增一个数"，就能让一个数自增。或许这一次这种额外的灵活性能让我们设计出拥有新能力的机器。

但这实际上只是又一个关于便利性而不是能力的特性。就像有限自动机实现正则表达式片段一样（参见 3.3.2 节），几个小图灵机可以连接在一起组成更大的图灵机，其中每一个小机器都实际上扮演着子例程的角色。我们之前看到的递增二进制数的机器，其状态和规则可构建入一个把两个二进制数相加的大一些的机器，而这个加法器本身还能构建成可执行乘法的更大的机器。

在小机器只需要由大机器的单个状态"调用"时，这很容易安排：只需要包含进小机器的副本，并把它的起始状态和接受状态与大机器的状态在子例程调用应该开始和结束的地方合并。这是我们使用递增机器组成一个加法器时期望的方式，因为规则手册的总体设计会根据需要重复单个任务——"如果第一个数不是 0，就递减第一个数并递增第二个数"。在机器中递增只需要发生在一个地方，而且在递增的工作完成之后只会有一个地方继续执行。

在我们想要在整个机器中的多个地方调用一个特定的子例程时，唯一的困难才会出现。一台图灵机没有办法存储"返回地址"，以让子例程知道一旦它结束之后应该返回到哪个状态，因此从表面上说，我们不能支持这种更通用的代码重用。但是就像在 5.3.1 节做的那样，可以用复制解决此问题：我们不是只构建较小机器状态和规则的一份副本，而是会构建出许多份，较大机器中需要使用的每一个地方都对应一份。

例如，把"递增一个数"的机器转换成"给一个数加三"的机器，最简单的方式是把三份副本连接到一起完成"递增一个数，然后递增一个数，再递增一个数"的总体设计。这通过几个中间状态跟踪通向最终目标的过程，其中每一个都从"递增这个数"发起，然后返回一个不同的中间状态。




```
>> def increment_rules(start_state, return_state)
     incrementing = start_state
```

```
    finishing = Object.new
    finished = return_state

    [
      TMRule.new(incrementing, '0', finishing,    '1', :right),
      TMRule.new(incrementing, '1', incrementing, '0', :left),
      TMRule.new(incrementing, '_', finishing,    '1', :right),
      TMRule.new(finishing,    '0', finishing,    '0', :right),
      TMRule.new(finishing,    '1', finishing,    '1', :right),
      TMRule.new(finishing,    '_', finished,     '_', :left)
    ]
  end
=> nil
>> added_zero, added_one, added_two, added_three = 0, 1, 2, 3
=> [0, 1, 2, 3]
>> rulebook = DTMRulebook.new(
    increment_rules(added_zero, added_one) +
    increment_rules(added_one, added_two) +
    increment_rules(added_two, added_three)
  )
=> #<struct DTMRulebook rules=[...]>
>> rulebook.rules.length
=> 18
>> tape = Tape.new(['1', '0', '1'], '1', [], '_')
=> #<Tape 101(1)>
>> dtm = DTM.new(TMConfiguration.new(added_zero, tape), [added_three], rulebook)
=> #<struct DTM ...>
>> dtm.run; dtm.current_configuration.tape
=> #<Tape 111(0)_>
```

只要我们能接受机器规模的扩张，用这种方式组合状态和规则的能力可以构建任意大小和复杂度的图灵机，无需任何对子例程的明确支持。

5.3.3 多纸带

有时候机器可以通过扩展它的外部存储提高能力。例如，在一台下推自动机可以访问第二个栈的时候，它会变得更强大，因为两个栈可以用来模拟一个无限纸带：每一个栈存储一半要模拟的纸带，而这台 PDA 可以在两个栈之间弹出和推入字符以模拟纸带头的动作，就像 5.1.4 节中的 `Tape` 实现那样。任何能访问无限纸带的有限状态机实际上都是一台图灵机，因此很明显增加一个额外的栈会让一台下推自动机更强大。

因此有理由期待通过增加一条或者多条纸带也能让图灵机更强大，这些纸带都有自己独立的纸带头。但事实又一次不是这样。一条图灵机的纸带通过交叉存取，会有足够的空间存储任意纸带数目的内容：包含 `abc`、`def` 和 `ghi` 的三条纸带可以一起存成 `adgbehcfi`。如果我们在每一个交叉字符的边上放上一个空白方格，机器就有地方指示所有模拟的纸带头的位置了：通过使用字符 X 指示每个纸带头的当前位置，我们可以用一条纸带 `a_dXg_b_e_hXcXf_i_` 表示纸带 `ab(c)`、`(d)ef` 和 `g(h)i` 的内容和纸带头的位置。

使用多条模拟的纸带对一台图灵机编程非常复杂，但累人的读、写以及纸带头的移动都可以封装成专门的状态和规则（“子例程”），这样机器的主要逻辑就不会变得过于复杂。在任何情况下，不管编程多么不方便，一台单纸带的图灵机最终都能执行多纸带机器能执行的任何任务，因此为一台图灵机增加额外的纸带并不会带来新的能力。

5.3.4　多维纸带

最后，尝试给一台图灵机更广阔的存储空间是很有诱惑力的。我们可以不使用线性纸带，而是提供无限的二维网格，并允许纸带头上下左右移动。每次需要移动纸带头快速访问外部存储的特定部分时，这都会很有用，而且不需要移动纸带头经过其他方格，这还允许我们在多个字符串周围留下无限的空白空间，这样它们中每一个都很容易变长，而不是每次在我们想要插入一个字符的时候只能手工整理整个纸带的信息以腾出空间来。

但不出意外的是，能用一维纸带模拟一个网格。最简单的方式就是使用两个一维纸带：主纸带实际存储数据，从纸带用来作为擦写空间。所模拟网格[4]的每一行都存储在主纸带上，顶上的行优先，并用一个特殊的字符标识每一行的结尾。

主纸带的头像往常一样位于当前字符，因此为了在模拟网格上左右移动，机器只是简单地左右移动纸带头。如果纸带头指向了行尾的标识符，就会用一个子例程整理纸带以便让网格扩展出一个空间。

为了在模拟网格中上下移动，纸带头必须向左或者向右分别移动完整的一行。机器会先移动纸带头到当前行的开头或者结尾，并使用从纸带记录移动的距离，然后把纸带头在前一行或者下一行移动同样的偏移量。如果纸带头离开了所模拟网格的最顶部或者最底部，可以使用一个子例程分配一个纸带头能移动进去的新空行。

这个模拟确实要求一台机器有两条纸带，但对此我们也知道如何模拟。这样最终把模拟的网格存储在两条模拟的纸带上，而这两条纸带本身存储在一条原始的纸带上。这两层模拟引入了大量的额外规则和状态，而且执行所模拟机器的一步就要花很多步，但规模的增加和速度的减慢并不妨碍它（最终）完成本来应该做的事情。

5.4　通用机器

尽管到目前为止我们看到的机器都有严重的缺陷：它们的规则都是硬编码的，这让它们无法适应不同的任务。一台能接受与一个特定正则表达式匹配的字符串的 DFA，不可能学会接受一个不同集合的字符串；一台能识别回文的 NPDA 将只能识别回文；一台递增二进制数的图灵机将永远不能做其他用途。

注 4：尽管网格本身是无限的，但只可能写入有限数目的字符，因此我们只需要存储包含所有空白字符的矩形区域即可。

大多数现实中的计算机不是这么工作的。现代计算机不是专门做某一项特殊工作的，而是为了*通用目的*而设计的并且能通过编程执行不同的任务。尽管一台可编程计算机的指令集和 CPU 设计是固定的，但能通过*软件*控制它的硬件并根据用户需要改变它的行为。

我们的简单机器能做这样的事情吗？在做一件不同的工作时，不必每次去设计一台新的机器，而是设计一台*简单*机器，它会从输入读取一个程序，然后做这个程序定义的任何工作。这办得到吗？

或许不足为奇的是，一台图灵机足够强大，它能从纸带读取一台*简单*机器的描述——比如说，一台确定性有限自动机——然后运行这台机器的模拟以找出它的工作内容。在 3.1.4 节，我们根据描述写下 Ruby 代码来模拟一台 DFA，现在只需要一点点工作就可以把那个代码的思想转化成一台图灵机的规则手册，以运行同样的模拟。



能模拟一台*特定* DFA 的图灵机和一台能模拟任何 DFA 的图灵机有着重要的区别。

设计一台图灵机重现一台特定 DFA 的行为很简单——毕竟，一台图灵机只不过是一台装有纸带的确定性有限自动机。DFA 规则手册的每一条规则都可以直接转成一个等价的图灵机规则；每一个转换过来的规则不是从 DFA 的外部输入流中读取，而是从纸带读取一个字符，并把纸带头移动到下一个方格。但这不是特别有趣，因为得到的图灵机并不比原始的 DFA 有用。

更有趣的是模拟*通用* DFA 的图灵机。这样的机器可以从纸带读取一个 DFA 的设计——规则、起始状态以及接受状态——然后遍历那台 DFA 执行的每一步，同时使用另一部分纸带跟踪模拟机器的当前状态和剩余的输入。通用模拟实现起来要难得多，但它让我们只要提供 DFA 的描述作为输入，就可以让图灵机做一台 DFA 能做的任何工作。

这也适用于对 NFA、DPDA 和 NPDA 的 Ruby 模拟，它们都可以转换成能模拟那种类型的任意自动机的一台图灵机。但关键是，对我们图灵机模拟本身，它也能起作用：通过把 `Tape`、`TMRule`、`DTMRulebook` 以及 `DTM` 重新实现成图灵机的规则，我们能设计一台图灵机，它能通过从纸带读取其规则、接受状态以及起始配置然后单步执行，模拟任何其他确定型图灵机，本质上这扮演着图灵机规则手册解释器的角色。完成这种工作的机器叫作*通用图灵机*（Universal Turing Machine，UTM）。

这非常激动人心，因为它在一个可编程装置中使图灵机的最大计算能力变得可用。我们可以把软件——经过编码的图灵机描述——写到纸带上，把这个纸带提供给 UTM，然后执行软件产生想要的行为。有限自动机和下推自动机不能用这种方式模拟它们自身的类型，因此图灵机不只标志着从能力有限的计算机器到能力强大的计算机器的过渡，还标志着从

单用途设备到全编程设备的转变。

简单地看一下一台通用图灵机如何工作。在实际构建一台 UTM 时，涉及大量的技巧和无趣的技术细节，因此我们的探索将会相当肤浅，但至少应该能证明这样的事情是可能的。

5.4.1 编码

在设计一台 UTM 的规则手册之前，我们得决定如何把一台完整的图灵机表示成纸带上的一个字符序列。一台 UTM 需要读取任意图灵机的规则、接受状态以及起始配置，然后随着模拟的进程，不断更新模拟机器的当前配置，因此我们需要一个实用的方式存储这些信息，以便 UTM 能与其协同工作。

有一个挑战，即每一台图灵机都只能在它的纸带上存储有限数目的状态和有限数目的不同字符，这两个数都由它的规则手册预先固定好了，当然 UTM 也不例外。如果我们设计一台 UTM，它能处理 10 个不同的纸带字符，那它如何模拟一台规则里使用 11 个字符的机器呢？如果我们更慷慨一些，让它能处理 100 个不同的字符，那么当想要模拟使用 1000 个字符的机器时会发生什么呢？不管我们为 UTM 自己的纸带设计多少个字符，为了直接表示每一个可能的图灵机它总是不够用的。

在所模拟机器和 UTM 之间还会有字符冲突的风险。为了在纸带上存储图灵机的规则和配置，我们需要能够用在 UTM 中有特殊含义的字符标注它们的边界，以便它能告诉我们从哪儿开始一个规则结束了，另一个规则开始了。但如果我们选择 `X` 作为规则之间的特定标识，则只要所模拟的任何一条规则中含有字符 X，都会有问题。即使我们设置一个保留字符的超级特殊集合，只给一台通用图灵机使用，如果试图模拟这台 UTM 本身的话仍然会引起问题，因此机器不会是真正通用的。这表明，我们需要某种转义，以避免所模拟机器的普通字符被 UTM 错误地解释成特殊字符。

我们可以解决这两个问题，方法是对所模拟机器的纸带内容使用固定指令系统的字符进行编码。如果编码体系只使用了特定的字符，那么我们可以保证对 UTM 来说把其他字符做特殊目的使用是安全的，而且如果这个体系能容纳任意数目的模拟状态和模拟字符，那就没有必要担心所模拟机器的规模和复杂度了。

只要能实现这些目标，这个编码体系的具体细节并不重要。举个例子，一个可能的方法是使用*一元*[5] 表示法把不同的值编码成同一字符重复不同次数的字符串：如果所模拟机器使用字符 `a`、`b` 和 `c`，它们可以编码成 `1`、`11` 和 `111`。另一个字符，如 `0`，可以用来作为值的分界标识：字符串 `abc` 可以标识成 `101110110111`。这种方法在空间上效率不是很高，但它可以通过在纸带上存储越来越长的由 `1` 组成的字符串来进行扩展，以容纳任意数目的编码的字符。

注 5：二元基于 2，一元基于 1。

一旦决定了如何对单个字符进行编码，我们就需要一种描述所模拟机器规则的方法。可以通过对规则的各个部分（状态、字符、下一状态、要写入的字符、移动方向）进行编码来实现，然后把它们在纸带上连接在一起，并在必要的地方使用特殊的分隔符。在示例的编码系统里，我们也可以用一元法表示状态——状态 1 是 `1`，状态 2 是 `11`，以此类推。但既然知道只会有两个方向，那我们可以使用任意的专用字符表示左和右（比如说 `L` 和 `R`）。

我们可以把单个的规则连到一起表示整个规则手册。类似地，可以通过把它当前状态的表示和它当前纸带内容的表示连在一起，来对所模拟机器的当前配置进行编码。[6] 而且这给了我们想要的：一台完整的图灵机以字符串的形式写在另一台图灵机的纸带上，准备通过模拟开始自己的生命周期。

5.4.2　模拟

从根本上说，通用图灵机和我们在 5.1.4 节构建的 Ruby 模拟的工作方式一样，只是要费力得多。

所模拟机器的描述——它的规则手册、接受状态以及起始配置——都以编码的格式存在于 UTM 的纸带上。为了执行模拟的一步，UTM 要在规则、当前状态和所模拟机器的纸带之间来回移动纸带头，以搜索出能应用到当前配置的一条规则。它找到一条规则的时候，就会根据规则里定义的字符和方向，更新所模拟的纸带，并把所模拟的机器放到新的状态上去。

这个过程会一直重复，直到所模拟的机器进入到一个接受状态，或者到达某个配置后因为没有规则应用处于卡死的状态。

注 6：我们没有详细说明纸带应该如何表示，但这也不难，而且总是可以选用 5.3.3 节的多纸带技术把它存储到所模拟的从纸带上。

第二部分

计算与可计算性

在本书的第一部分，我们已经讨论了几个熟悉的计算示例：命令式编程语言、状态机，以及通用计算机。那些示例向我们展示了计算差不多就是使用一个系统操纵信息并回答问题的过程。

在第二部分，我们将会大胆些，先在不熟悉的地方寻求计算，最后探索关于计算机器所能做之事的根本限制。

作为程序员，我们与编程语言和机器打交道，它们是根据我们对世界的认知模型进行设计的，而且我们期望它们带有一些特性，能轻松地把我们的思想转换成实现。这些以人为中心的设计是由便利性而非必要性驱动的，甚至一台设计简单的图灵机，也会让我们想起用纸和铅笔工作的数学家。

但是计算并不只会发生在友好的、为我们所熟悉的机器上。更多不寻常的系统的计算能力同样强大，即使它们内部的工作机制对于人类来说不容易控制或理解。我们将探索这个思想，在第 6 章尝试用极小的语言（这种语言似乎根本没什么有用的特性）写程序，并在第 7 章审视各种简单的系统，看看它们如何像更复杂的机器一样执行同样的计算。

在确信许多种系统里都可能发生强大的计算后，第 8 章将探讨计算本身的能力。人们很自然地认为，只要付出足够的时间和努力写一个合适的程序，就能让计算机解决几乎任何问题，但事实证明了存在一个理论约束：有些问题无法用任何计算机解决，不管它多快多高效。

遗憾的是，一些不能解决的问题涉及程序行为的预测，而这恰好是程序员想要计算机帮他们做的。我们将会看到一些应对计算世界中这些硬限制的策略，而第 9 章将探索如何利用抽象找出无法回答的问题的近似答案。

第6章

从零开始编程

> 如果你想从头开始制作苹果派，必须先创造整个宇宙。
>
> ——卡尔 · 萨根

本书中，我们一直在试图构建计算模型来理解计算。到目前为止，我们设计了想象中带有不同约束的简单机器，并看到不同的约束会产生出拥有不同计算能力的系统，以此对计算进行了建模。

第 5 章的图灵机很有意思，因为它们不依赖复杂的特性就能实现复杂的行为。只要有一条纸带、一个读写头以及一个固定的规则集合，图灵机就足以模拟拥有更好存储能力、支持非确定性执行或者任何其他奇妙特性的机器行为。这告诉我们，成熟的计算不需要机器具备大量的潜在复杂性，只需要其具备存储、检索以及使用数据进行简单决策的能力。

计算模型不一定非要看起来像机器，它们可以看起来像编程语言。第 2 章的 Simple 编程语言当然可以执行计算，但它的执行过程没有图灵机那么优雅。它已经有了大量语法（数字、布尔值、二进制表达式、变量、赋值、序列、条件、循环），而且我们甚至还没有开始为其增加特性，以使其适合写真正的程序：字符串、数据结构、过程调用，等等。

把 Simple 转换成真正有用的编程语言将会是一项艰苦的工作，最终的设计会包含大量的细节，不会对揭示计算的本质帮助太多。从零开始创建某个最小的东西——编程语言世界的一台图灵机，这样我们就可以看到对于计算来说，哪些特性是本质的，哪些特性是偶然的噪音。

本章，我们将研究一种叫作*无类型 lambda 演算*（untyped lambda calculus）的极小编程语

言。首先，我们将用尽可能少的语言特性写（用 Ruby）一些接近 lambda 演算的程序。这将仍然仅仅是在用 Ruby 编程，但施加虚构的约束之后，我们便能很轻松地探索一个受限的语义，而不需要学习一门新语言。然后，我们了解到这些非常有限的特性集合能做什么以后，就将利用这些特性把它们实现为一种语言（使用它自己的解析器、抽象语法和操作语义）——使用我们在之前章节中学到的技术。

6.1 模拟lambda演算

为了理解如何使用最小语言编程，我们不打算使用 Ruby 诸多有用的特性来解决问题。很自然，这意味着没有 gem，没有标准库，没有模块（module）、方法、类或者对象，既然我们试图尽可能地做到最小，那还将避免使用控制结构、赋值、数组、字符串、数字和布尔值。

当然，如果我们避免使用 Ruby 的所有特性，那就没有语言可用来编程了，因此下面是将要保留的：

- 对变量进行引用；
- 创建 proc；
- 调用 proc。

这意味着只能写出如下样子的 Ruby 代码：

```
-> x { -> y { x.call(y) } }
```



这大致就是无类型 lambda 演算程序的样子，足以接近我们的目的了。6.2 节会详细讨论 lambda 演算。

为了让代码更简短并且更容易阅读，我们还将使用常量作为缩写：如果创建了一个复杂的表达式，可以把它赋值给一个常量，给它一个短名字以便以后再次使用。引用这个名字与重新输入原始表达式没有区别（名字只是让代码更加简洁），因此我们会依赖于 Ruby 的赋值特性。任意时刻都可以通过替换每一个常量所引用的 proc 来取消缩写，这样做的代价是会让程序变得更长。

6.1.1 使用proc工作

既然要用 proc 构建整个程序，让我们在深度使用它们之前花一分钟看看它们的属性。



目前，我们将使用完整特性的 Ruby 来描绘 proc 的一般行为。在我们开始写代码来解决 6.12 节的“问题”时，才会施加这些限制。

1. 管道

proc 是值在程序中进行移动的管道。考虑调用下面的 proc 时会发生什么：

```
-> x { x + 2 }.call(1)
```

作为参数提供给调用的值 `1`，传入代码块 `x` 的参数中，然后把参数传给用到它的所有地方，因此 Ruby 最后会对 `1+2` 求值。语言的其他部分会做实际的工作，proc 只是把一部分程序连接在一起并让值流向需要它的地方。

对使用最小化 Ruby 的实验来说这已经有了不好的兆头。如果 proc 只能在实际使用值的 Ruby 片段之间移动值，那怎么才能只用 proc 就能构建有用的程序呢？探索完 proc 的其他属性之后，我们就会理解。

2. 参数

proc 可以带有多个参数，但这不是一个本质特性。如果得到一个能处理多个参数的 proc……

```
-> x, y {
  x + y
}.call(3, 4)
```

……我们总是可以将其重写为嵌入式的单参数 proc：

```
-> x {
  -> y {
    x + y
  }
}.call(3).call(4)
```

这里，外部 proc 的参数是 `x`，而且会返回内部的 proc，内部的 proc 也带有一个参数 `y`。我们可以使用 x 的一个值调用外部的 proc，然后使用 `y` 的一个值调用内部的 proc，而且我们会得到与多参数时同样的结果。[1]

既然我们在尽可能多地去掉 Ruby 的特性，那就限制自己只创建和调用单参数的 proc 吧。这不会让事情变得更糟糕。

3. 等价

查明一个 proc 内部代码的唯一途径就是调用它，因此如果使用同样的参数调用两个 proc，会产生相同结果的话，那么即使它们的内部代码不同，它们也是可交换的。这种根据外部可见行为判断两者相等的思想叫作*外延等价*（extensional equality）。

比如说我们有一个叫 `p` 的 proc：

注 1：这叫作 curry 化，并且我们可以使用 `Proc#curry` 自动进行这个转换。

```
>> p = -> n { n * 2 }
=> #<Proc (lambda)>
```

我们可以再创建一个叫 `q` 的 proc，它带有一个参数并且只是用这个参数调用 `p`：

```
>> q = -> x { p.call(x) }
=> #<Proc (lambda)>
```

`p` 和 `q` 明显是两个不同的 proc，但它们外延相等，因为它们对任何参数来讲都会做同样的事情：

```
>> p.call(5)
=> 10
>> q.call(5)
=> 10
```

知道 `p` 与 `-> x { p.call(x) }` 等价，这就为重构提供了新的机会。如果在我们的程序里看到 `-> x { p.call(x) }` 这种一般模式，我们可以选择用 `p` 替换整个表达式来消除它，而在某些情况下（后面会看到），我们可能会决定采用相反的方式。

4. 语法

对于创建和调用 proc，Ruby 提供了一个语法选择。从现在开始，我们会使用 `->` *`arguments`* `{` *`body`* `}` 创建一个 proc，然后使用方括号调用它：

```
>> -> x { x + 5 }[6]
=> 11
```

这样无需额外的语法就很容易看到 proc 的主体和参数。

6.1.2　问题

我们的目标是写出著名的 FizzBuzz 程序：

> 写一个程序输出数字 1 到 100。但如果数字是 3 的倍数，就不输出数字而是输出“Fizz”，如果是 5 的倍数就输出“Buzz”。对于那些 3 和 5 的公倍数，就输出“FizzBuzz”。
>
> —— Imran Ghory，“用 FizzBuzz 找到热爱编码的开发者”（Using FizzBuzz to Find Developers who Grok Coding，http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/）

这是故意挑选的一个简单问题，用来测试一个面试者是否有编程经验。任何知道如何编程的人应该都能毫无困难地解决这个问题。

下面是使用完整特性 Ruby 的一个实现：

```
(1..100).each do |n|
```

```
    if (n % 15).zero?
      puts 'FizzBuzz'
    elsif (n % 3).zero?
      puts 'Fizz'
    elsif (n % 5).zero?
      puts 'Buzz'
    else
      puts n.to_s
    end
  end
```

这不是 FizzBuzz 最聪明的一个实现（还存在着大量更聪明的实现，http://redd.it/10d7w），但它很直接，任何人都可以不用思考就写出来。

但是，这个程序含有一些 puts 语句，而我们没法只使用 proc 就把文本输出到控制台，[2] 因此我们把它替换成一个大致等价的程序，这个程序只返回一个字符串数组而不是输出它们：

```
(1..100).map do |n|
  if (n % 15).zero?
    'FizzBuzz'
  elsif (n % 3).zero?
    'Fizz'
  elsif (n % 5).zero?
    'Buzz'
  else
    n.to_s
  end
end
```

对 FizzBuzz 问题来说这仍然是一个有意义的解决方案，但现在的这个版本我们有可能只用 proc 就实现了。

不管它多简单，如果没有一种编程语言的任何特性的话，这仍然是要求非常高的程序：它创建一个范围，对其做映射，对一个大的条件求值，使用取模操作进行算数运算，使用 Fixnum#zero? 预测，使用一些字符串，而且还用 Fixnum#to_s 把数字转换成字符串。这用到了很多 Ruby 内建功能，而我们将要把它们全部去除再用 proc 重新实现。

6.1.3 数字

我们准备从关注 FizzBuzz 中出现的数字开始。怎么才能不用 Fixnum 或者 Ruby 提供的任何其他数据类型，就表示出数字呢？

如果打算从头开始实现数字 [3]，我们最好对要实现的东西有个透彻的理解。到底什么是数字呢？如果不对试图定义的东西的某个方面进行假设，就很难给出一个具体的定义。例

注 2：我们当然可以对向控制台输出进行建模，引入一个 proc 来表示标准输出，然后设计如何向它发送文本，但那会让我们的练习复杂化，而且变得没意思。FizzBuzz 不是关于输出的，而是关于算数和控制流的。

注 3：具体说来，我们这里想要实现的是非负整数：0、1、2、3 等。

如，“某个东西告诉我们有多少……”没有用，因为“多少”只是“数字”的另一种表述方式。

下面是描绘数字特征的一种方式：想象我们有一袋子苹果和一袋子橘子。我们从一个袋子里取出一个苹果，从另一个袋子里取出一个橘子，然后把它们放到一起。之后我们不断地取出一个苹果和一个橘子，直到至少其中有一个袋子变成空的。

如果两个袋子同时变成空的，我们就学到了一件有趣的事情：尽管含有不同的东西，但这两个袋子有一个共有的属性，这个属性意味着它们同时变空了；在不断从每个袋子里取出水果的每一个时刻，两个袋子都不是空的或者两个袋子都是空的。袋子共有的这个抽象性质就是我们可以叫作数字的东西（尽管不知道是哪个数字！），而且我们可以把这两个袋子与世界上的任何其他袋子做比较，来看看跟它们是不是有着同样的“数”。

因此描绘数字特征的一种方式是某个动作的重复（或者叫*迭代*），在这个例子中动作是从袋子里取一个物体。每一个数字都与重复一个动作的唯一方式对应：数字 1 对应的是只执行这个动作；数字 2 对应的是执行这个动作然后再次执行；以此类推。并不奇怪，数字 0 对应着根本不执行这个动作。

既然创建和调用 proc 是这里程序唯一可以执行的“动作”，我们可以尝试用代码实现一个数字 `n`，在代码里对调用 proc 这个动作重复 `n` 次。

例如，如果允许定义方法——这是不允许的，不过我们只是玩一玩——那么我们可以把 `#one` 定义成一个方法，它带有一个 proc 参数以及另一个任意的参数，而且它会用该任意参数调用 proc：

```
def one(proc, x)
  proc[x]
end
```

我们还可以定义 `#two`，它会调用一次 proc，然后用第一次调用的结果对其再次调用：[4]

```
def two(proc, x)
  proc[proc[x]]
end
```

以此类推：

```
def three(proc, x)
  proc[proc[proc[x]]]
end
```

按照这种模式，可以很自然地把 `#zero` 定义为一个带有 proc 和另一个参数的方法，这个方法完全忽略 proc（换句话说，对其调用零次），并且会原封不动地返回第二个参数：

注 4：这叫作“迭代这个函数”。

```
def zero(proc, x)
  x
end
```

所有这些实现都可以转换成无方法的表示。例如，我们可以用带有两个参数[5]的 proc 替换方法 #one，然后用第二个调用参数调用第一个参数。它们看起来是这样：

```
ZERO  = -> p { -> x {       x    } }
ONE   = -> p { -> x {     p[x]   } }
TWO   = -> p { -> x {   p[p[x]]  } }
THREE = -> p { -> x { p[p[p[x]]] } }
```

这避免了不允许我们使用的功能，而且通过把它们赋值给常量还给了 proc 名字。



把数据表示为纯代码的技术称为邱奇编码（Church encoding），它是以 lambda 演算（http://dx.doi.org/10.2307/2371045）的发明者阿隆佐·邱奇的名字命名的。这些数字是邱奇数（Church numeral），而且我们很快将会看到邱奇布尔值（Church Boolean）和邱奇有序对（Church pair）的例子。

尽管在 FizzBuzz 解决方案里我们回避了 Ruby 的特性，但是一旦超出了我们的代码范围，把数字的这些外部表示转换成 Ruby 值会很有用，这样它们就能在控制台进行检查和在测试中断言，或者至少能让我们相信它们确实本来代表数字。

幸运的是，可以写一个 #to_integer 方法执行这个转换：

```
def to_integer(proc)
  proc[-> n { n + 1 }][0]
end
```

这个方法带有表示一个数字的 proc 并用另一个 proc 和原始的 Ruby 数字 0 来调用它（这个 proc 只是递增它的参数）。如果我们使用 ZERO 调用 #to_integer，那么因为 ZERO 的定义，递增的 proc 不会得到调用，这样我们会原封不动得到 0：

```
>> to_integer(ZERO)
=> 0
```

而如果用 THREE 调用 #to_integer，递增的 proc 将会被调用三次，这样我们得到 Ruby 的 3：

```
>> to_integer(THREE)
=> 3
```

因此基于 proc 的表示只是在对数字进行编码，并且我们可以根据需要把它们转成更实用的表示。

注 5：实际上，“带有两个参数”并不准确，因为我们已经限制自己只使用单参数的 proc 了（参见 6.1.1 节中“参数”部分）。准确的说法是“带有一个参数并且返回一个带有另一个参数的新的 proc”，但那太绕嘴了，所以我们采用这种简略的说法，只是要记住真正的意思是什么。

对于 FizzBuzz，需要数字 5、15 和 100，它们都可以用同样的技术实现：

```
FIVE    = -> p { -> x { p[p[p[p[p[x]]]]] } }
FIFTEEN = -> p { -> x { p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]] } }
HUNDRED = -> p { -> x { p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
]]]]]]]]]]]]]]]]]]]]]]] } }
```

这些都不是很简洁的定义，但它们确实可以工作，就像用 #to_integer 确认的那样：

```
>> to_integer(FIVE)
=> 5
>> to_integer(FIFTEEN)
=> 15
>> to_integer(HUNDRED)
=> 100
```

因此，回到 FizzBuzz 程序，所有的 Ruby 数字都可以用基于 proc 的实现替换：

```
(ONE..HUNDRED).map do |n|
  if (n % FIFTEEN).zero?
    'FizzBuzz'
  elsif (n % THREE).zero?
    'Fizz'
  elsif (n % FIVE).zero?
    'Buzz'
  else
    n.to_s
  end
end
```



我们写成 ONE 而不是 -> p { -> x { p[x] } } 等，这是为了让代码更清晰。

遗憾的是，这个程序不再工作了，因为我们在对基于 proc 的数字实现上使用了像 .. 和 % 这样的运算符。因为不知道如何处理，所以 Ruby 将会这样报错：TypeError: can't iterate from Proc，NoMethodError: undefined method `%' for #<Proc (lambda)>。为了使用这些表示，我们需要替换掉所有运算，并且只能使用 proc 完成。

但是在我们能重新实现任何一个操作之前，需要实现 true 和 false。

6.1.4 布尔值

我们怎样才能只用 proc 表示布尔值呢？布尔值只会存在于条件语句当中，而且通常情况下，一个条件会说"if 某个布尔值 then 这样 else 那样"：

```
>> success = true
=> true
>> if success then 'happy' else 'sad' end
=> "happy"
>> success = false
=> false
>> if success then 'happy' else 'sad' end
=> "sad"
```

所以一个布尔值的真正工作是允许在两个选项中做选择，因此我们可以利用这一点，把布尔值表示成在两个值中选择其一的 proc。我们不是把一个布尔值看成一段无生命的代码，它被将来的代码读取并能决定选择两个选项中的哪一个，而只是直接把它实现为一段代码，这段代码在用两个选项进行调用的时候，要么选择第一个选项要么选择第二个。

实现成方法的 #true 和 #false 可能是：

```
def true(x, y)
  x
end

def false(x, y)
  y
end
```

#true 是一个带有两个参数并返回第一个参数的方法，而 #false 带有两个参数并返回第二个。这足够提供给我们粗线条的条件行为了：

```
>> success = :true
=> :true
>> send(success, 'happy', 'sad')
=> "happy"
>> success = :false
=> :false
>> send(success, 'happy', 'sad')
=> "sad"
```

像以前一样直接把这些方法转换成 proc：

```
TRUE  = -> x { -> y { x } }
FALSE = -> x { -> y { y } }
```

就像之前定义了 #to_integer 方法作为检查，以便能够把基于 proc 的数字转换成 Ruby 数字一样，我们可以定义 #to_boolean 方法，以便能把 TRUE 和 FALSE 的 proc 转换成 Ruby 原始的 true 和 false 对象：

```
def to_boolean(proc)
  proc[true][false]
end
```

这个函数带有一个表示布尔值的参数，然后使用 true 作为第一个参数而 false 作为第二个参数调用它。TRUE 只是会返回它的第一个参数，因此 to_boolean（TRUE）将会返回 true，而 FALSE 会返回 false：

```
>> to_boolean(TRUE)
=> true
>> to_boolean(FALSE)
=> false
```

因此用 proc 表示布尔值出奇地简单，但对于 FizzBuzz，我们不只需要布尔值，还需要用 proc 实现 Ruby 的 if-elseif-else。事实上，由于这些布尔值实现的工作方式，很容易写出 #if 方法：

```
def if(proc, x, y)
  proc[x][y]
end
```

而这很容易转换成一个 proc：

```
IF =
  -> b {
    -> x {
      -> y {
        b[x][y]
      }
    }
  }
```

很明显 IF 不需要做什么有用的工作，因为布尔值自己就会找到合适的参数——IF 只是添加的糖——但看起来比直接调用布尔值更自然：

```
>> IF[TRUE]['happy']['sad']
=> "happy"
>> IF[FALSE]['happy']['sad']
=> "sad"
```

这还意味着我们可以修改 #to_boolean 方法以使用 IF：

```
def to_boolean(proc)
  IF[proc][true][false]
end
```

尽管我们在重构，但值得一提的是，像 6.1.1 节中“相等”部分讨论的那样，IF 的实现含有与更简单的 proc 等价的 proc，所以 IF 的实现能被显著简化。例如看一下 IF 的最内层实现：

```
-> y {
  b[x][y]
}
```

这段代码的意思是：

(1) 带上一个参数 y；
(2) 用参数 x 调用 b 得到一个 proc；
(3) 用参数 y 调用这个 proc。

第 (1) 步和第 (3) 步没什么用：在我们使用一个参数调用这个 proc 的时候，它只是把这个参数传给另一个 proc。因此整个 proc 只是与第 (2) 步等价，也就是 b[x]，而我们可以把无用的代码从 IF 的实现中移除，以便让它更简洁：

```
IF =
  -> b {
    -> x {
      b[x]
    }
  }
```

在最内层我们又看到了同样的模式：

```
-> x {
  b[x]
}
```

基于同样的原因，这个 proc 与 b 相同，因此我们可以进一步简化 IF：

```
IF = -> b { b }
```

我们不能再进一步简化了。



IF 没做什么有用的事情（是 TRUE 和 FALSE 在做全部的工作），因此我们可以去掉它以做进一步的简化。但我们的目标是把原始的 FizzBuzz 程序尽可能忠实地转换成 proc，因此尽管 IF 仅仅起到装饰作用，但使用 IF 提醒我们 if-elsif-else 表达式在原始程序中出现的位置会很方便。

不管怎样，现在有了 IF，可以回到 FizzBuzz 程序把 Ruby 的 if-elsif-else 替换成对 IF 的嵌套调用了：

```
(ONE..HUNDRED).map do |n|
  IF[(n % FIFTEEN).zero?][
    'FizzBuzz'
  ][IF[(n % THREE).zero?][
    'Fizz'
  ][IF[(n % FIVE).zero?][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

6.1.5 谓词

我们下一步的工作是用基于 proc 的实现替换 `Fixnum#zero?`，这个实现将会与基于 proc 的数字一起工作。处理 Ruby 值的 `#zero?` 的基本算法像下面这样：

```
def zero?(n)
  if n == 0
    true
  else
    false
  end
end
```

（这有些冗余，但它明确了所发生的事情：把这个数字与 0 比较；如果相等就返回 `true`，否则返回 `false`。）

我们如何才能让它处理 proc 而不是 Ruby 数字呢？请再看一下数字的实现：

```
ZERO  = -> p { -> x {       x    } }
ONE   = -> p { -> x {     p[x]   } }
TWO   = -> p { -> x {   p[p[x]]  } }
THREE = -> p { -> x { p[p[p[x]]] } }
⋮
```

注意，`ZERO` 是唯一不调用 `p` 的数字——它只是返回 `x`——但所有其他的数字至少会调用 `p` 一次。我们可以利用这一点：如果用 `TRUE` 作为第二个参数调用一个未知的数字，则如果数字是 `ZERO`，它将立即返回 `TRUE`。如果不是 `ZERO`，它会返回调用 `p` 返回的东西，因此如果我们让 `p` 成为一个总是返回 `FALSE` 的 proc，就会得到想要的行为：

```
def zero?(proc)
  proc[-> x { FALSE }][TRUE]
end
```

把它重写成一个 proc 还是很容易：

```
IS_ZERO = -> n { n[-> x { FALSE }][TRUE] }
```

我们可以使用 `#to_boolean` 在控制台上检查它的工作情况：

```
>> to_boolean(IS_ZERO[ZERO])
=> true
>> to_boolean(IS_ZERO[THREE])
=> false
```

这工作得很好，所以在 FizzBuzz 里，我们可以把所有对 `#zero?` 的调用替换成 `IS_ZERO`：

```
(ONE..HUNDRED).map do |n|
  IF[IS_ZERO[n % FIFTEEN]][
```

```
    'FizzBuzz'
  ][IF[IS_ZERO[n % THREE]][
    'Fizz'
  ][IF[IS_ZERO[n % FIVE]][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

6.1.6　有序对

我们已经有了数字和布尔值形式的可用数据，但还没有能有条理地存储超过一个值的任何数据结构。为了实现更复杂的功能，我们将很快需要某种数据结构，因此先来介绍一个。

最简单的数据结构是*有序对*（pair），它跟二元数组类似。有序对实现起来非常容易：

```
PAIR  = -> x { -> y { -> f { f[x][y] } } }
LEFT  = -> p { p[-> x { -> y { x } } ] }
RIGHT = -> p { p[-> x { -> y { y } } ] }
```

一个有序对的作用是存储两个值，并在之后根据需要再次提供。为了构建一个有序对，我们用两个值（一个 x 和一个 y）调用 PAIR，然后返回它的内部 proc：

```
-> f { f[x][y] }
```

这个 proc 在用另一个为 f 的 proc 调用时，会用较早的 x 和 y 的值作为参数回调它。LEFT 和 RIGHT 会从一个有序对中分别选出左边和右边的元素，它们会调用一个 proc，这个 proc 分别返回其第一个和第二个参数。它足够简单：

```
>> my_pair = PAIR[THREE][FIVE]
=> #<Proc (lambda)>
>> to_integer(LEFT[my_pair])
=> 3
>> to_integer(RIGHT[my_pair])
=> 5
```

这个非常简单的数据结构足够我们使用了；6.1.8 节中将使用有序对，将其作为更复杂结构的一个基础结构。

6.1.7　数值运算

现在有了数字、布尔值、条件、谓词以及有序对，我们几乎准备好重新实现模运算符了。

在对两个数进行模运算之前，我们需要能够执行更简单的运算，如递增和递减一个数。递增相当直接：

```
INCREMENT = -> n { -> p { -> x { p[n[p][x]] } } }
```

看一下 INCREMENT 如何工作：我们用基于 proc 的数字 n 调用它，它会返回一个新的 proc，这个 proc 像数字那样带有某个其他 proc p 和某个任意的第二参数 x。

我们调用这个新的 proc 的时候它会做什么呢？首先它会以 p 和 x 作为参数调用 n——因为 n 是一个数字，所以这意味着就像原始的数字那样，“在 x 上对 p 进行 n 次调用”——然后对结果再调用一次 p。那么总体说来，这个 proc 的第一个参数会在它的第二个参数上调用 n+1 次，这恰好是表示数字 n+1 的方法。

但递减呢？这看起来是个更难的问题：一旦一个 proc 已经调用了 n 次，再额外增加一次调用以便成为 n+1 次调用是相当容易的，但没有明显的方法可以撤销一次调用以便成为 n-1 次调用。

一个解决办法就是设计一个 proc，在对某个初始参数调用 n 次的时候返回数字 n-1。幸运的是，有序对正好可以帮助我们实现这种方法。思考一下这个 Ruby 方法所做的：

```
def slide(pair)
  [pair.last, pair.last + 1]
end
```

在我们用数字组成的二元数组为参数调用 slide 时，它会返回一个新的二元数组，这个二元数组包含第二个数字还有比第二个数字大 1 的数字；如果输入的数组包含的是连续数字，那么效果就是向上“滑动”一个数字窗口：

```
>> slide([3, 4])
=> [4, 5]
>> slide([8, 9])
=> [9, 10]
```

这很有用，因为通过在 -1 处开始一个窗口，我们可以安排一种情况，让数组里的第一个数字比我们调用 slide 的次数小 1，即使我们只是在递增数据：

```
>> slide([-1, 0])
=> [0, 1]
>> slide(slide([-1, 0]))
=> [1, 2]
>> slide(slide(slide([-1, 0])))
=> [2, 3]
>> slide(slide(slide(slide([-1, 0]))))
=> [3, 4]
```

我们不能只用基于 proc 的数字完成，因为没法表示 -1，但 side 的有趣之处是不管怎样它只关注数组中的第二个数，因此我们可以放入任意的哑值（dummy value）——比如说 0——替换掉 -1，这样仍然能得到同样的结果：

```
>> slide([0, 0])
=> [0, 1]
>> slide(slide([0, 0]))
=> [1, 2]
>> slide(slide(slide([0, 0])))
=> [2, 3]
>> slide(slide(slide(slide([0, 0]))))
=> [3, 4]
```

这是让 DECREMENT 工作的关键：我们可以把 slide 转成一个 proc，使用数字 n 的 proc 表示对由 ZERO 组成的有序对调用 slide n 次，然后使用 LEFT 从结果的有序对中拉出左边的数来：

```
SLIDE     = -> p { PAIR[RIGHT[p]][INCREMENT[RIGHT[p]]] }
DECREMENT = -> n { LEFT[n[SLIDE][PAIR[ZERO][ZERO]]] }
```

下面是 DECREMENT 的作用：

```
>> to_integer(DECREMENT[FIVE])
=> 4
>> to_integer(DECREMENT[FIFTEEN])
=> 14
>> to_integer(DECREMENT[HUNDRED])
=> 99
>> to_integer(DECREMENT[ZERO])
=> 0
```



DECREMENT[ZERO] 的结果实际上只是最初的 PAIR[ZERO][ZERO] 值的左边元素，在这种情况下根本就没有对其调用过 SLIDE。既然没有负值，0 就是我们能提供给 DECREMENT[ZERO] 的最合理的答案，因此使用 0 作为哑值是个好主意。

既然我们有了 INCREMENT 和 DECREMENT，就可能实现类似加法、减法、乘法和取幂这样的数字运算了：

```
ADD      = -> m { -> n { n[INCREMENT][m] } }
SUBTRACT = -> m { -> n { n[DECREMENT][m] } }
MULTIPLY = -> m { -> n { n[ADD[m]][ZERO] } }
POWER    = -> m { -> n { n[MULTIPLY[m]][ONE] } }
```

这些实现在很大程度上是自解释的。如果我们想要 m 加 n，只需要“从 m 开始对其递增 n 次”，同样这也适用于减法；有了 ADD 之后，我们可以进行 m 乘 n，方法是“从 ZERO 开始，对其进行 n 次 ADD m”，使用 MULTIPLY 和 ONE 进行幂运算也类似。



在 6.2.2 节“规约表达式”部分中，我们将用 Ruby 完成 ADD[ONE][ONE] 的小步求值，以便展示它如何产生 TWO。

这些算数足够我们起步了，但在能用 proc 实现 % 之前，我们需要了解一个执行模运算的

算法。下面是其对 Ruby 数字的处理：

```
def mod(m, n)
  if n <= m
    mod(m - n, n)
  else
    m
  end
end
```

例如，为了计算 17 模 5 可以进行如下操作：

- 如果 5 小于等于 17，这是事实，那么就用 17 减去 5，然后在结果上调用 #mod 方法，也就是说 12 模 5；
- 5 小于等于 12，因此尝试 7 模 5；
- 5 小于等于 7，因此尝试 2 模 5；
- 5 不再小于等于 2，因此返回结果 2。

但我们还不能用 proc 实现 #mod，因为它使用了另一个运算符 <=，我们还没有实现它，因此需要暂时先用 proc 实现 <=。

可以从看起来不相干的对 Ruby 数的 #less_or_equal? 实现开始：

```
def less_or_equal?(m, n)
  m - n <= 0
end
```

这没什么用，因为它依赖于 <=，但至少它把问题分解成了两个我们已经解决的其他问题了：减法和与零作比较。减法我们已经处理过了，与零的相等性我们也完成了，但我们如何实现小于等于零的判断呢？

碰巧我们不需要担心，因为零已经是我们知道如何实现的最小的数了。回忆一下，我们基于 proc 的数字都是非负的，因此“小于零”在我们的数字系统里是无意义的概念。

如果从一个小一点的数里用 SUBSTRACT 减去一个大一点的数，将只会返回 ZERO，因为没法返回一个负数，并且 ZERO 是能得到的最接近的值了[6]：

```
>> to_integer(SUBTRACT[FIVE][THREE])
=> 2
>> to_integer(SUBTRACT[THREE][FIVE])
=> 0
```

我们已经写了 IS_ZERO，并且因为如果 m 小于等于 n（也就是说 n 至少与 m 一样大）的话 SUBTRACT[m][n] 会返回 ZERO，所以足可以用 proc 实现 #less_or_equal? 了：

注 6：你可能会抗议 3-5=0 不叫“减法”，你是对的：这种运算的专业名称叫“monus”，因为加法之下的非负整数形成的是可交换幺半群而不是一个合适的阿贝尔群。

```
def less_or_equal?(m, n)
  IS_ZERO[SUBTRACT[m][n]]
end
```

让我们把这个方法转成 proc：

```
IS_LESS_OR_EQUAL =
  -> m { -> n {
    IS_ZERO[SUBTRACT[m][n]]
  } }
```

它能正常工作吗？

```
>> to_boolean(IS_LESS_OR_EQUAL[ONE][TWO])
=> true
>> to_boolean(IS_LESS_OR_EQUAL[TWO][TWO])
=> true
>> to_boolean(IS_LESS_OR_EQUAL[THREE][TWO])
=> false
```

看起来不错。

这补上了 #mod 实现中缺少的部分，因此可以用 proc 重写它：

```
def mod(m, n)
  IF[IS_LESS_OR_EQUAL[n][m]][
    mod(SUBTRACT[m][n], n)
  ][
    m
  ]
end
```

并用一个 proc 替换掉方法定义：

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      MOD[SUBTRACT[m][n]][n]
    ][
      m
    ]
  } }
```

太好了！它能工作吗？

```
>> to_integer(MOD[THREE][TWO])
SystemStackError: stack level too deep
```

不能。

Ruby 在调用 MOD 的时候进入了无限递归循环，因为我们把 Ruby 的原始功能转换成 proc 时

漏掉了条件语义中一些重要的东西。在像 Ruby 这样的语言里，`if-else` 语句是非严格的（或者说是懒的）：我们给它一个条件和两个代码块，然后它会对条件求值以决定对哪个代码块求值并返回——它从来也不会对两个代码块都求值。

`IF` 实现的问题是我们无法利用构建到 Ruby 的 `if-else` 里的懒性行为。我们只能说“调用一个 proc，`IF`，其参数是两个其他的 proc”，因此 Ruby 冲出来，在 `IF` 有机会决定返回哪个之前就对两个参数都进行求值。

再看一下 `MOD`：

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      MOD[SUBTRACT[m][n]][n]
    ][
      m
    ]
  } }
```

在我们对 `m` 和 `n` 调用 `MOD`，而 Ruby 开始对内部 proc 的代码体求值时，它会对 `MOD[SUBTRACT[m][n]][n]` 进行递归调用并立即开始把它当作传递给 `IF` 的参数求值，不管 `IS_LESS_OR_EQUAL[n][m]` 是 `TRUE` 还是 `FALSE`。对 `MOD` 第二次调用的结果是又一次无条件的递归调用，以此类推，从而会无限递归下去。

为了修正，我们需要一种方式告诉 Ruby 延迟对 `IF` 第二个参数的求值，直到确定需要对其求值为止。Ruby 中任何表达式的求值都可以通过封装到一个 proc 里延迟，但在一个 proc 内封装一个任意的 Ruby 值通常会改变其含义（如 `1+2` 的结果并不等于 `->{1+2}`），因此我们可能需要做得更聪明一些。

幸运的是没必要这样做，因为这是一个特殊情况：我们知道因为所有的值都是单参数的 proc，所以调用 `MOD` 的结果也将会是一个单参数的 proc，并且我们已经知道（参见 6.1.1 节中“相等”部分），对于任意的 proc `p`，另一个 proc 将其封装，它与 `p` 参数相同并立即用此参数调用 `p`，它们将会产生同样的值，因此我们可以使用这个技巧延迟递归调用而不影响传递给 `IF` 的值的含义：

```
MOD =
  -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        MOD[SUBTRACT[m][n]][n][x]
      }
    ][
      m
    ]
  } }
```

这把递归的 `MOD` 调用封装到 `-> x { ...[x] }` 以对其延迟。Ruby 现在不会在调用 `IF` 的时候试图对这个 proc 的代码体求值了，但如果这个 proc 被 IF 选中并作为结果返回，它就能被接受者调用，最终触发（现在肯定是需要的）对 `MOD` 的递归调用。

`MOD` 现在能工作吗？

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
     POWER[THREE][THREE]
   ][
     ADD[THREE][TWO]
   ])
=> 2
```

是的，太好啦！

但是先别庆祝，因为还有一个更棘手的问题：我们在用常量 `MOD` 定义常量 `MOD`，因此这个定义不只是一个缩写。这次我们不仅仅在把一个复杂的 proc 赋值给一个常量以便之后重用。事实上，我们在依赖 Ruby 的赋值语义，尽管仍然在定义 `MOD`，但它很明显还没有被定义，然而我们可以在 `MOD` 的实现中引用它，并期望在之后对其求值的时候它已经被定义了。

那是在欺骗，因为原则上我们应该能撤销掉所有的缩写——“我们提到 `MOD` 的地方，实际的意思是这个长长的 proc”——但只要 `MOD` 由其自身定义这就不可能。

我们可以使用 `Y` 组合子解决此问题，这些著名的辅助代码恰恰是为此目的：无欺骗地定义一个递归函数。下面是它的样子：

```
Y = -> f { -> x { f[x[x]] }[-> x { f[x[x]] }] }
```

三言两语很难解释 Y 组合子，但下面是一个梗概（技术上不准确）：当我们使用一个 proc 调用 Y 组合子的时候，它会用 proc *本身作为第一个参数*对 proc 进行调用。因此，如果我们写了一个需要一个参数的 proc 并用那个 proc 调用这个 Y 组合子，那么这个 proc 将会把自身作为参数，从而只要它想要调用自身的时候就可以使用那个参数。

悲剧的是，由于和 `MOD` 永远循环一样的原因，Y 组合子在 Ruby 中也会永远循环下去，因此我们需要一个修订后的版本。是表达式 `x[x]` 引起了这个问题，而我们可以再次修正这个问题，方法是每次这个表达式出现，就把它封装到 `-> y { ...[y] }` 内部以延迟它们的求值：

```
Z = -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }
```

这是 Z *组合子*，它是 Y 组合子对于像 Ruby 这样严格语言的变换。

最后我们可以创建 `MOD` 的一个满意实现了，方法是给 `MOD` 提供一个额外的参数 `f`，封装对

围绕它的 Z 组合子的调用，这样在我们之前调用 MOD 的地方都可以调用 f：

```
MOD =
  Z[-> f { -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        f[SUBTRACT[m][n]][n][x]
      }
    ][
      m
    ]
  } } }]
```

谢天谢地，MOD 的这个无欺骗的版本仍然能工作：

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
     POWER[THREE][THREE]
   ][
     ADD[THREE][TWO]
   ])
=> 2
```

现在我们可以把 FizzBuzz 程序中 % 出现的地方都替换成 MOD 的调用：

```
(ONE..HUNDRED).map do |n|
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

6.1.8　列表

对于 FizzBuzz 我们只遗留了几个 Ruby 特性要重新实现：范围（range）、#map、字符串字面量以及 Fixnum#to_s。对于已经实现的值和运算我们已经看到了大量细节，因此我们将会快速浏览其余的特性并尽可能地减少细节。（不要担心需要理解所有的东西，我们只是浅尝辄止。）

为了能够实现范围和 #map，我们需要实现列表（list），而构建列表的最简单方法就是使用有序对（pair）。这个实现像链表一样工作，其中每个有序对都保存一个值和一个指向链表中下一个有序对的指针。在这里，我们不使用指针而是使用嵌入式的有序对。标准的列表运算看起来是这样：

```
EMPTY     = PAIR[TRUE][TRUE]
UNSHIFT   = -> l { -> x {
              PAIR[FALSE][PAIR[x][l]]
            } }
IS_EMPTY  = LEFT
FIRST     = -> l { LEFT[RIGHT[l]] }
REST      = -> l { RIGHT[RIGHT[l]] }
```

它们像这样工作：

```
>> my_list =
     UNSHIFT[
       UNSHIFT[
         UNSHIFT[EMPTY][THREE]
       ][TWO]
     ][ONE]
=> #<Proc (lambda)>
>> to_integer(FIRST[my_list])
=> 1
>> to_integer(FIRST[REST[my_list]])
=> 2
>> to_integer(FIRST[REST[REST[my_list]]])
=> 3
>> to_boolean(IS_EMPTY[my_list])
=> false
>> to_boolean(IS_EMPTY[EMPTY])
=> true
```

使用 `FIRST` 和 `REST` 取出列表中的单个元素相当笨拙，因此就像处理数字和布尔值那样，我们可以写一个 `#to_array` 方法以便在控制台上提供帮助：

```
def to_array(proc)
  array = []

  until to_boolean(IS_EMPTY[proc])
    array.push(FIRST[proc])
    proc = REST[proc]
  end

  array
end
```

这让监视列表更为容易：

```
>> to_array(my_list)
=> [#<Proc (lambda)>, #<Proc (lambda)>, #<Proc (lambda)>]
>> to_array(my_list).map { |p| to_integer(p) }
=> [1, 2, 3]
```

如何实现范围呢？事实上，与其找到一种方式显式地把范围表示成 proc，不如只写一个 proc，它可以构建范围内的所有元素的列表。对于原始的 Ruby 数字和“列表”（如数组），我们可以这么写：

```
def range(m, n)
  if m <= n
    range(m + 1, n).unshift(m)
  else
    []
  end
end
```

在预期可用的列表操作方面，这个算法稍嫌做作，但能讲得通：由 m 到 n 所有数字组成的列表与由 m+1 到 n 组成的列表（并在前头放上 m）一样；如果 m 比 n 大，那这个由数字组成的列表就是空的。

幸运的是，我们已经有了把这个方法直接转换成 proc 所需要的一切：

```
RANGE =
  Z[-> f {
    -> m { -> n {
      IF[IS_LESS_OR_EQUAL[m][n]][
        -> x {
          UNSHIFT[f[INCREMENT[m]][n]][m][x]
        }
      ][
        EMPTY
      ]
    } }
  }]
```



注意 Z 组合子对递归的使用，以及条件语句的 TRUE 分支周围的 `-> x { ... [x] }`。

它能正常工作吗?

```
>> my_range = RANGE[ONE][FIVE]
=> #<Proc (lambda)>
>> to_array(my_range).map { |p| to_integer(p) }
=> [1, 2, 3, 4, 5]
```

是的，可以正常工作，所以让我们在 FizzBuzz 中使用：

```
RANGE[ONE][HUNDRED].map do |n|
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    'Buzz'
  ][
    n.to_s
  ]]]
end
```

为了实现 #map，我们可以使用一个叫 FOLD 的辅助方法，它有点像 Ruby 中的 Enumerable#inject：

```
FOLD =
  Z[-> f {
    -> l { -> x { -> g {
      IF[IS_EMPTY[l]][
        x
      ][
        -> y {
          g[f[REST[l]][x][g]][FIRST[l]][y]
        }
      ]
    } } }
  }]
```

FOLD 令写出能处理列表中每一项元素的 proc 变得更简单：

```
>> to_integer(FOLD[RANGE[ONE][FIVE]][ZERO][ADD])
=> 15
>> to_integer(FOLD[RANGE[ONE][FIVE]][ONE][MULTIPLY])
=> 120
```

一旦有了 FOLD，我们就可以简洁地写出 MAP 来：

```
MAP =
  -> k { -> f {
    FOLD[k][EMPTY][
      -> l { -> x { UNSHIFT[l][f[x]] } }
    ]
  } }
```

MAP 能正常工作吗？

```
>> my_list = MAP[RANGE[ONE][FIVE]][INCREMENT]
=> #<Proc (lambda)>
>> to_array(my_list).map { |p| to_integer(p) }
=> [2, 3, 4, 5, 6]
```

是的，可以正常工作。因此我们可以替换掉 FizzB 中的 #map 了：

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    'FizzBuzz'
  ][IF[IS_ZERO[MOD[n][THREE]]][
    'Fizz'
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    'Buzz'
  ][
    n.to_s
  ]]]
}]
```

差不多完成了！就剩下处理字符串了。

6.1.9 字符串

字符串很容易处理：我们可以只是把它们表示成由数字组成的列表，只要对哪个数字表示哪个字符的编码达成一致就可以。

我们可以选择任何编码，因此不使用像 ASCII 这样的通用目的的编码，而是设计一种对于 FizzBuzz 更方便的新型编码。只需要对数字和字符串 `'FizzBuzz'`、`'Fizz'` 以及 `'Buzz'` 进行编码就可以，因此可以使用 `0` 到 `9` 表示字符 `'0'` 到 `'9'`，而把字符 `'B'`、`'F'`、`'i'`、`'u'` 和 `'z'` 编码成 `10` ～ `14`。

这样我们就有了一种方式来表示需要的字符串字面量（注意不要截断 Z 组合子）：

```
TEN = MULTIPLY[TWO][FIVE]
B   = TEN
F   = INCREMENT[B]
I   = INCREMENT[F]
U   = INCREMENT[I]
ZED = INCREMENT[U]

FIZZ     = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[EMPTY][ZED]][ZED]][I]][F]
BUZZ     = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[EMPTY][ZED]][ZED]][U]][B]
FIZZBUZZ = UNSHIFT[UNSHIFT[UNSHIFT[UNSHIFT[BUZZ][ZED]][ZED]][I]][F]
```

为了检查其是否能正常工作，可以写一些外部的方法，把它们转换成 Ruby 字符串：

```
def to_char(c)
  '0123456789BFiuz'.slice(to_integer(c))
end

def to_string(s)
  to_array(s).map { |c| to_char(c) }.join
end
```

好了，字符串能工作了吗？

```
>> to_char(ZED)
=> "z"
>> to_string(FIZZBUZZ)
=> "FizzBuzz"
```

太好啦。那么可以在 FizzBuzz 中使用它们了：

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    FIZZBUZZ
  ][IF[IS_ZERO[MOD[n][THREE]]][
    FIZZ
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    BUZZ
  ][
```

```
    n.to_s
  ]]]
}]
```

最后要实现的是 `Fixnum#to_s`。为此，我们需要能把数分割成组成它的数字，下面是一种用 Ruby 实现的方法：

```
def to_digits(n)
  previous_digits =
    if n < 10
      []
    else
      to_digits(n / 10)
    end

  previous_digits.push(n % 10)
end
```

还没有实现 `<`，但可以通过使用 `n <= 9` 而不是 `n < 10` 来规避这个问题。遗憾的是，我们没法回避实现 `Fixnum#/` 和 `Array#push`，下面是它们的实现：

```
DIV =
  Z[-> f { -> m { -> n {
    IF[IS_LESS_OR_EQUAL[n][m]][
      -> x {
        INCREMENT[f[SUBTRACT[m][n]][n]][x]
      }
    ][
      ZERO
    ]
  } } }]
PUSH =
  -> l {
    -> x {
      FOLD[l][UNSHIFT[EMPTY][x]][UNSHIFT]
    }
  }
```

现在可以把 `#to_digits` 转换成一个 proc 了：

```
TO_DIGITS =
  Z[-> f { -> n { PUSH[
    IF[IS_LESS_OR_EQUAL[n][DECREMENT[TEN]]][
      EMPTY
    ][
      -> x {
        f[DIV[n][TEN]][x]
      }
    ]
  ][MOD[n][TEN]] } }]
```

它能工作吗？

```
>> to_array(TO_DIGITS[FIVE]).map { |p| to_integer(p) }
=> [5]
>> to_array(TO_DIGITS[POWER[FIVE][THREE]]).map { |p| to_integer(p) }
=> [1, 2, 5]
```

是的，可以工作。而且因为我们已经预见性地设计了一种字符串编码，在这种字符串编码里，`1` 代表 `'1'`，以此类推，所以由 `TO_DIGITS` 产生的数组已经是有效的字符串了：

```
>> to_string(TO_DIGITS[FIVE])
=> "5"
>> to_string(TO_DIGITS[POWER[FIVE][THREE]])
=> "125"
```

因此我们可以在 FizzBuzz 中用 `TO_DIGITS` 替换 `#to_s`：

```
MAP[RANGE[ONE][HUNDRED]][-> n {
  IF[IS_ZERO[MOD[n][FIFTEEN]]][
    FIZZBUZZ
  ][IF[IS_ZERO[MOD[n][THREE]]][
    FIZZ
  ][IF[IS_ZERO[MOD[n][FIVE]]][
    BUZZ
  ][
    TO_DIGITS[n]
  ]]]
}]
```

6.1.10　解决方案

我们最终完成了！（这可能是有史以来最长的、最笨拙的工作面试了。）现在我们已经有了完全由 proc 写成的 FizzBuzz 的实现。来运行一下以确保它正常工作：

```
>> solution =
     MAP[RANGE[ONE][HUNDRED]][-> n {
       IF[IS_ZERO[MOD[n][FIFTEEN]]][
         FIZZBUZZ
       ][IF[IS_ZERO[MOD[n][THREE]]][
         FIZZ
       ][IF[IS_ZERO[MOD[n][FIVE]]][
         BUZZ
       ][
         TO_DIGITS[n]
       ]]]
     }]
=> #<Proc (lambda)>
>> to_array(solution).each do |p|
     puts to_string(p)
   end; nil
1
2
Fizz
```

```
4
Buzz
Fizz
7
.
.
.
94
Buzz
Fizz
97
98
Fizz
Buzz
=> nil
```

经历了这么多麻烦以确保每一个常量只是某个更长表达式的一个缩写，我们认为有必要把每一个常量用它的定义替换，因此可以看到完整的程序了：

```
-> k { -> f { -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }
[-> f { -> l { -> x { -> g { -> b { b }[-> p { p[-> x { -> y { x } }] }[l]][x]
[-> y { g[f[-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y { y } }] }
[l]] }[l]][x][g]][-> l { -> p { p[-> x { -> y { x } }] }[-> p { p[-> x { -> y
{ y } }] }[l]] }[l]][y] }] } } } }][k][-> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { x } }][-> x { -> y { x } }]][-> l { -> x { -> l { -> x { -> x { -> y
{ -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[l][f[x]] } }] } }[-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y
{ x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-> x
{ -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-
> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x
{ -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { ->
y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-
> p { -> x { x } }]]] }][m] } }[m][n]] } }[m][n]][-> x { -> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[f[-> n { -> p { -> x { p[n[p][x]] } } }[m]][n]][m][x] }][-> x { ->
y { -> f { f[x][y] } } }[-> x { -> y { x } }][-> x { -> y { x } }]] } } }][-> p
{ -> x { p[x] } }][-> p { -> x
{ p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[
p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]
]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]] } }]][->
n { -> b { b }[-> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-
> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> m
{ -> n { -> b { b }[-> m { -> n { -> n { n[-> x { -> x { -> y { y } } }][-> x
{ -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p
{ -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n
{ -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x
{ -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p { -> x { x } }]]] }]
[m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> p { -> x
{ p[p[p[p[p[p[p[p[p[p[p[p[p[p[p[x]]]]]]]]]]]]]]] } }]]][-> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
```

```
[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }]
[-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f
{ f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { ->
y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { ->
y { -> f { f[x][y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { x } }][-> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -
> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { ->
x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> m { -> n { n[-> m { -
> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }
[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> n { -> p
{ -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -
> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-
> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m
{ -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-
> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p
{ -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x
{ p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }]
[-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]][-> n { -> p { -> x { p[n[p][x]] } } }
[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }
[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]]][-> b { b }[-> n { n[-> x { -> x { -> y { y } } }][-
> x { -> y { x } }] }[-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x]
[y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-> x { -> x
{ -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p { p[-> x
{ -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -
> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-> n { -
> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p
{ p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-
> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x
{ x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> p { -
> x { p[p[p[x]]] } }]]]][-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { ->
x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
```

```
[y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -
> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x
{ -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x]
[y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x { -> y { x } }][-
> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p]
[x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p
{ -> x { p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -
> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { ->
x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -
> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> b { b }[-> n { n[-> x
{ -> x { -> y { y } } }][-> x { -> y { x } }] }[-> f { -> x { f[-> y { x[x]
[y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m
{ -> n { -> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -
> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x]
[y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }
[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[->
p { -> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-
> m { -> n { n[-> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -
> f { f[x][y] } } }[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> p { p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f
{ f[x][y] } } }[-> p { -> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n]
[x] }][m] } } }][n][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]][-> l { -> x { -> x { -
> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }
[x][l]] } }[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }]
[-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-> l { -> x { -> x { -> y { -> f
{ f[x][y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
> y { -> f { f[x][y] } } }[x][l]] } }[-> x { -> y { -> f { f[x][y] } } }[-> x
{ -> y { x } }][-> x { -> y { x } }]][-> n { -> p { -> x { p[n[p][x]] } } }[->
n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n
{ -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -
> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]][-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x
{ p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m
{ -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x
{ x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]]]
[-> n { -> p { -> x { p[n[p][x]] } } }[-> n { -> p { -> x { p[n[p][x]] } } }[-
> n { -> p { -> x { p[n[p][x]] } } }[-> m { -> n { n[-> m { -> n { n[-> n { ->
p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x
{ p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]]]]][-> m { -> n { n[-> m { -
> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-> p { -> x { x } }] } }
[-> p { -> x { p[p[x]] } }][-> p { -> x { p[p[p[p[p[x]]]]] } }]]][-> f { -> x
{ f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f { -> n { -> l { -
> x { -> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-> y { x[x][y] }] }] }[-> f
{ -> l { -> x { -> g { -> b { b }[-> p { p[-> x { -> y { x } }] }[l]][x][-> y
```

```
{ g[f[-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y { y } }] }
[l]] }[l]][x][g]][-> l { -> p { p[-> x { -> y { y } }] }[-> p { p[-> x { -> y
{ y } }] }[l]] }[l]][y] }] } } } }][l][-> l { -> x { -> x { -> y { -> f { f[x]
[y] } } }[-> x { -> y { y } }][-> x { -> y { -> f { f[x][y] } } }[x][l]] } }[-
> x { -> y { -> f { f[x][y] } } }[-> x { -> y { x } }][-> x { -> y { x } }]][x]]
[-> l { -> x { -> x { -> y { -> f { f[x][y] } } }[-> x { -> y { y } }][-> x { -
> y { -> f { f[x][y] } } }[x][l]] } }] } }[-> b { b }[-> m { -> n { -> n { n[-
> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p
{ p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-
> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x
{ -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }]
[-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }[-> m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p]
[x]] } } }][m] } }[m]][-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p
{ -> x { p[p[p[p[p[x]]]]] } }]]]][-> x { -> y { -> f { f[x][y] } } }[-> x { ->
y { x } }][-> x { -> y { x } }]][-> x { f[-> f { -> x { f[-> y { x[x][y] }] }[-
> x { f[-> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -
> n { n[-> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-
> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }
[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p
{ p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -
> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { -> n { -
> p { -> x { p[n[p][x]] } } }[f[-> m { -> n { n[-> n { -> p { p[-> x { -> y
{ x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-> x { -> y
{ y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x { -> y
{ y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }][-> p
{ -> x { x } }]]] }][m] } }[m][n]][n]][x] }][-> p { -> x { x } }] } } }][n][->
m { -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]]
[-> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]][x] }]][-> f { -> x { f[-> y { x[x][y] }] }[-> x { f[-
> y { x[x][y] }] }] }[-> f { -> m { -> n { -> b { b }[-> m { -> n { -> n { n[-
> x { -> x { -> y { y } } }][-> x { -> y { x } }] }[-> m { -> n { n[-> n { -> p
{ p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }[-> p { p[-
> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p { p[-> x
{ -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -> x { x } }]
[-> p { -> x { x } }]]] }][m] } }[m][n]] } }[n][m]][-> x { f[-> m { -> n { n[-
> n { -> p { p[-> x { -> y { x } }] }[n[-> p { -> x { -> y { -> f { f[x][y] } } }
[-> p { p[-> x { -> y { y } }] }[p]][-> n { -> p { -> x { p[n[p][x]] } } }[-> p
{ p[-> x { -> y { y } }] }[p]]] }][-> x { -> y { -> f { f[x][y] } } }[-> p { -
> x { x } }][-> p { -> x { x } }]]] }][m] } }[m][n]][n][x] }][m] } } }][n][-> m
{ -> n { n[-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[m]][-
> p { -> x { x } }] } }[-> p { -> x { p[p[x]] } }][-> p { -> x
{ p[p[p[p[p[x]]]]] } }]]] } }][n]]]] }]
```

太漂亮了。

6.1.11　高级编程技术

构建完全由 proc 组成的程序需要很多努力，但我们已经明白只要不介意应用一些技巧，完成实际工作是可能的。来快速看一下用这个最小环境写代码的其他几个技术。

1. 无限流

使用代码表示数据有一些有趣的优点。我们基于 proc 的列表不一定是静态的：列表也是代码，在我们传递它给 FIRST 和 REST 时它能做正确的事情，因此很容易实现能动态计算自身内容的列表，也就是流（stream）。事实上，流没有理由是有限的，因为计算只需要根据需要生成列表的内容就可以了，所以它可以一直无限产生新的值。

例如，下面是一个零组成的无限流的实现：

```
ZEROS = Z[-> f { UNSHIFT[f][ZERO] }]
```



这是 ZEROS = UNSHIFT[ZEROS][ZERO] 的“无欺骗”版本，即用它自身定义的数据结构。作为一个程序员，我们通常会觉得用自身定义一个递归函数的思想很舒服，但用自身定义一个数据结构看起来很怪异；在这种情况下，它们几乎是同样的东西，而 Z 组合子让两者都完全合理了。

在控制台上，我们可以看到 ZEROS 表现得就像一个列表，尽管这个列表看不到尽头：

```
>> to_integer(FIRST[ZEROS])
=> 0
>> to_integer(FIRST[REST[ZEROS]])
=> 0
>> to_integer(FIRST[REST[REST[REST[REST[REST[ZEROS]]]]]])
=> 0
```

能有一个辅助方法把这个流转成一个 Ruby 的数组会很方便，但 to_array 会永远运行下去，直到我们明确地让这个转换进程停下来为止。一个可选的“最大数”的参数可以做到这一点：

```
def to_array(l, count = nil)
  array = []

  until to_boolean(IS_EMPTY[l]) || count == 0
    array.push(FIRST[l])
    l = REST[l]
    count = count - 1 unless count.nil?
  end

  array
end
```

这让我们可以从流中获取任意数目的元素并把它们转成一个数组：

```
>> to_array(ZEROS, 5).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0]
>> to_array(ZEROS, 10).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>> to_array(ZEROS, 20).map { |p| to_integer(p) }
=> [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

ZEROS 不会每次都对一个新的元素进行计算，但做起来也非常简单。下面是一个从给定值累加的流：

```
>> UPWARDS_OF = Z[-> f { -> n { UNSHIFT[-> x { f[INCREMENT[n]][x] }][n] } }]
=> #<Proc (lambda)>
>> to_array(UPWARDS_OF[ZERO], 5).map { |p| to_integer(p) }
=> [0, 1, 2, 3, 4]
>> to_array(UPWARDS_OF[FIFTEEN], 20).map { |p| to_integer(p) }
=> [15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34]
```

下面是一个包含一个给定数字所有倍数的流：

```
>> MULTIPLES_OF =
     -> m {
       Z[-> f {
         -> n { UNSHIFT[-> x { f[ADD[m][n]][x] }][n] }
       }][m]
     }
=> #<Proc (lambda)>
>> to_array(MULTIPLES_OF[TWO], 10).map { |p| to_integer(p) }
=> [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
>> to_array(MULTIPLES_OF[FIVE], 20).map { |p| to_integer(p) }
=> [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100]
```

我们可以像其他列表一样操纵这些无限流。例如，可以通过对已有的 proc 映射一个新的 proc 得到一个新的流：

```
>> to_array(MULTIPLES_OF[THREE], 10).map { |p| to_integer(p) }
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>> to_array(MAP[MULTIPLES_OF[THREE]][INCREMENT], 10).map { |p| to_integer(p) }
=> [4, 7, 10, 13, 16, 19, 22, 25, 28, 31]
>> to_array(MAP[MULTIPLES_OF[THREE]][MULTIPLY[TWO]], 10).map { |p| to_integer(p) }
=> [6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
```

甚至可以写一个 proc 把两个流组合成第三个流：

```
>> MULTIPLY_STREAMS =
     Z[-> f {
       -> k { -> l {
         UNSHIFT[-> x { f[REST[k]][REST[l]][x] }][MULTIPLY[FIRST[k]][FIRST[l]]]
       } }
     }]
=> #<Proc (lambda)>
>> to_array(MULTIPLY_STREAMS[UPWARDS_OF[ONE]][MULTIPLES_OF[THREE]], 10).
     map { |p| to_integer(p) }
=> [3, 12, 27, 48, 75, 108, 147, 192, 243, 300]
```

因为流的内容能由任何计算生成，所以我们创建斐波那契数列的无限列表，或者质数，或者按字母顺序的所有可能的字符串，或者任何其他可计算的东西都已经没有障碍了。这个抽象非常强大，除了已有的特性之外不需要任何智能的特性了。

原始 Ruby 流

Ruby 有一个 Enumerator 类可以用来构建无限的流，而不需要依赖 proc。下面是“给定数的倍数”的流的实现方法：

```
def multiples_of(n)
  Enumerator.new do |yielder|
    value = n
    loop do
      yielder.yield(value)
      value = value + n
    end
  end
end
```

这个方法返回一个 Enumerator，每次我们对其调用 #next，它都会执行 loop 的一个迭代并返回获得的值：

```
>> multiples_of_three = multiples_of(3)
=> #<Enumerator: #<Enumerator::Generator>:each>
>> multiples_of_three.next
=> 3
>> multiples_of_three.next
=> 6
>> multiples_of_three.next
=> 9
```

Enumerator 类包括了 Enumerable 模块，因此我们可以调用 #first、#take 和 #detect 这样的方法：

```
>> multiples_of(3).first
=> 3
>> multiples_of(3).take(10)
=> [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
>> multiples_of(3).detect { |x| x > 100 }
=> 102
```

其他的 Enumerable 方法，如 #map 和 #select，在这个 Enumerator 上没法正常工作，因为它们会尝试处理这个无限流中的每一项。但是，Ruby 2.0 的 Enumerator::Lazy 类重新实现了一些 Enumerable 方法，这样它们在依赖的 Enumerator 继续计数时仍然可以工作。我们可以通过在一个 Enumerator 上调用 #lazy 来获得一个 Enumerator::Lazy，然后可以像之前操纵 proc 版本一样操纵这些无限流：

```
>> multiples_of(3).lazy.map { |x| x * 2 }.take(10).force
=> [6, 12, 18, 24, 30, 36, 42, 48, 54, 60]
>> multiples_of(3).lazy.map { |x| x * 2 }.select { |x| x > 100 }.take(10).force
=> [102, 108, 114, 120, 126, 132, 138, 144, 150, 156]
>> multiples_of(3).lazy.zip(multiples_of(4)).map { |a, b| a * b }.take(10).force
=> [12, 48, 108, 192, 300, 432, 588, 768, 972, 1200]
```

与基于 proc 的列表相比，这不是很整洁（为了处理无限流，我们得写一些特殊的代码，而不能只是像通常的 Enumerable 那样处理），但它表明 Ruby 确实含有处理这些不寻常数据结构的内建方式。

2. 避免随意递归

在 FizzBuzz 练习里，我们使用 MOD 和 RANGE 这样的递归函数展示了 Z 组合子的用法。这很方便，因为它让我们从一个没有约束的递归的 Ruby 实现转换成一个基于 proc 的实现，而不必改变代码结构，但是从技术上讲，没有 Z 组合子我们也可以利用邱奇数的行为来实现这些函数。

例如，MOD[m][n] 的实现方法是，只要 n<=m 就不断地从 m 中减去 n，并且总是检查这个条件以决定是否进行下一次的递归调用。但如果只是对"如果 n <= m 就从 m 中减去 n"这个动作执行固定的次数，而不是使用递归动态控制这个重复的过程，也可以得到同样的结果。我们不知道需要重复的确切次数，但知道 m 次肯定够了（最差情况就是 n 为 1），而且多做几次也无碍：

```
def decrease(m, n)
  if n <= m
    m - n
  else
    m
  end
end

>> decrease(17, 5)
=> 12
>> decrease(decrease(17, 5), 5)
=> 7
>> decrease(decrease(decrease(17, 5), 5), 5)
=> 2
>> decrease(decrease(decrease(decrease(17, 5), 5), 5), 5)
=> 2
>> decrease(decrease(decrease(decrease(decrease(17, 5), 5), 5), 5), 5)
=> 2
```

因此我们可以重写 MOD 以利用一个 proc，这个 proc 的参数是一个数，它或是从这个数中减去 m（如果它比 n 大）或是直接返回这个数。这个 proc 对 m 本身调用 m 次，以便获得最终的答案：

```
MOD =
  -> m { -> n {
    m[-> x {
      IF[IS_LESS_OR_EQUAL[n][x]][
        SUBTRACT[x][n]
      ][
        x
      ]
    }][m]
  } }
```

MOD 的这个版本与递归版本工作得一样出色：

```
>> to_integer(MOD[THREE][TWO])
=> 1
>> to_integer(MOD[
     POWER[THREE][THREE]
   ][
     ADD[THREE][TWO]
   ])
=> 2
```

尽管这个实现比原来的实现简单，但它不仅难以阅读而且通常效率更低，因为它总是会执行重复调用的最差情况下的次数而不是尽可能早地停下来。在外延上它也与原来的实现不等价，因为老版本的 MOD 如果被要求除零的话会永远循环下去（条件 n<=m 永远不会为 false），而这个实现只是返回它的第一个参数：

```
>> to_integer(MOD[THREE][ZERO])
=> 3
```

RANGE 更有挑战一些，但我们可以使用与让 DECREMENT 工作时类似的技巧：设计一个函数，在对某个初始参数调用 n 次时，它会从预想的范围里返回 n 个数的列表。就像 DECREMENT 一样，秘诀是使用一个有序对存储结果的列表和在下一个迭代中需要的信息：

```
def countdown(pair)
  [pair.first.unshift(pair.last), pair.last - 1]
end

>> countdown([[], 10])
=> [[10], 9]
>> countdown(countdown([[], 10]))
=> [[9, 10], 8]
>> countdown(countdown(countdown([[], 10])))
=> [[8, 9, 10], 7]
>> countdown(countdown(countdown(countdown([[], 10]))))
=> [[7, 8, 9, 10], 6]
```

重写 proc 很容易：

```
COUNTDOWN = -> p { PAIR[UNSHIFT[LEFT[p]][RIGHT[p]]][DECREMENT[RIGHT[p]]] }
```

现在我们只需要实现 RANGE 以便它调用 COUNTDOWN 正确的次数（从 m 到 n 的范围内总是有 m-n+1 个元素）并从最终的有序对中取出结果列表：

```
RANGE = -> m { -> n { LEFT[INCREMENT[SUBTRACT[n][m]][COUNTDOWN][PAIR[EMPTY][n]]] } }
```

这个无组合子的版本工作得也很好：

```
>> to_array(RANGE[FIVE][TEN]).map { |p| to_integer(p) }
=> [5, 6, 7, 8, 9, 10]
```



可以通过执行事先决定好次数的迭代来实现 MOD 和 RANGE——而不是执行一个会一直运行直到条件变为 true 才停止的任意的循环——因为它们是*原始递归函数*。参见 7.2 节可以了解更多内容。

6.2 实现lambda演算

FizzBuzz 实现已经让我们对用无类型的 lambda 演算写程序有了一些感觉。这些限制迫使我们从零开始实现大量的基本功能而不是依赖语言的特性，但我们确实成功构建了解决这个问题所需要的数据结构和算法。

因为还没有 lambda 演算的解释器，所以还没有*真正*写演算的程序呢。我们只是在用 lambda 演算的形式写 Ruby 程序，以此获得这样一个小语言能工作的感觉。但我们已经有了构建 lambda 演算解释器并用其对实际的 lambda 演算表达式求值的所有知识，那来尝试一下吧。

6.2.1 语法

无类型的 lambda 表达式是一种编程语言，它只有三种表达式：变量、函数定义以及调用。我们不再引入一种新的 lambda 表达式语法，而是还遵守 Ruby 的习惯（变量看起来像 x，函数看起来像 ->x{x}，而调用看起来像是 x[y]），并尽量不让两种语言混淆。



为什么是“lambda 演算”？

在这个上下文中，单词*演算*（calculus）的意思是一个操纵符号字符串的规则系统。[7] lambda 演算的原始语法用的是希腊字母 lambda（λ）代替 Ruby 中的 -> 符号。例如，ONE 会写成 λp.λx.p x。

我们可以用常见的方式实现 LCVariable、LCFunction 和 LCCall 类：

```
class LCVariable < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
      to_s
  end
end

class LCFunction < Struct.new(:parameter, :body)
  def to_s
```

注 7：大多数人把它与*微积分学*联系起来，这是一个数学函数中关于改变率和数量累加率的系统。

```
    "-> #{parameter} { #{body} }"
  end

  def inspect
    to_s
  end
end

class LCCall < Struct.new(:left, :right)
  def to_s
    "#{left}[#{right}]"
  end

  def inspect
    to_s
  end
end
```

这些类可以让我们构建 lambda 演算表达式的抽象语法树，就像第 2 章的 Simple 和第 3 章的正则表达式那样：

```
>> one =
     LCFunction.new(:p,
       LCFunction.new(:x,
         LCCall.new(LCVariable.new(:p), LCVariable.new(:x))
       )
     )
=> -> p { -> x { p[x] } }
>> increment =
     LCFunction.new(:n,
       LCFunction.new(:p,
         LCFunction.new(:x,
           LCCall.new(
             LCVariable.new(:p),
             LCCall.new(
               LCCall.new(LCVariable.new(:n), LCVariable.new(:p)),
               LCVariable.new(:x)
             )
           )
         )
       )
     )
=> -> n { -> p { -> x { p[n[p][x]] } } }
>> add =
     LCFunction.new(:m,
       LCFunction.new(:n,
         LCCall.new(LCCall.new(LCVariable.new(:n), increment), LCVariable.new(:m))
       )
     )
=> -> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }
```

因为这种语言有这样小的语法，所以那三个类足以表示任意的 lambda 演算的程序了。

6.2.2　语义

现在通过为每个语法类实现一个 #reduce 方法来为 lambda 演算赋予一个小步操作语义。小步操作语义是一个很有吸引力的选择，因为它能让我们看到求值的每一步，这在 Ruby 表达式中是没法轻易做到的。

1. 替换变量

在实现 #reduce 之前，我们需要另一个叫作 #replace 的操作，它能找到一个表达式里的一个特定变量并用另一个表达式替换它：

```
class LCVariable
  def replace(name, replacement)
    if self.name == name
      replacement
    else
      self
    end
  end
end

class LCFunction
  def replace(name, replacement)
    if parameter == name
      self
    else
      LCFunction.new(parameter, body.replace(name, replacement))
    end
  end
end

class LCCall
  def replace(name, replacement)
    LCCall.new(left.replace(name, replacement), right.replace(name, replacement))
  end
end
```

对于变量和调用，它的工作方式很明显：

```
>> expression = LCVariable.new(:x)
=> x
>> expression.replace(:x, LCFunction.new(:y, LCVariable.new(:y)))
=> -> y { y }
>> expression.replace(:z, LCFunction.new(:y, LCVariable.new(:y)))
=> x
>> expression =
     LCCall.new(
       LCCall.new(
         LCCall.new(
           LCVariable.new(:a),
           LCVariable.new(:b)
         ),
```

```
          LCVariable.new(:c)
        ),
        LCVariable.new(:b)
      )
=> a[b][c][b]
>> expression.replace(:a, LCVariable.new(:x))
=> x[b][c][b]
>> expression.replace(:b, LCFunction.new(:x, LCVariable.new(:x)))
=> a[-> x { x }][c][-> x { x }]
```

对于函数，情况会更复杂。#replace 只能对一个函数的函数体起作用，而且它只能替换自由变量——自由变量就是在函数范围内但是没有被声明为函数参数的变量：

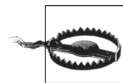
```
>> expression =
     LCFunction.new(:y,
       LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
     )
=> -> y { x[y] }
>> expression.replace(:x, LCVariable.new(:z))
=> -> y { z[y] }
>> expression.replace(:y, LCVariable.new(:z))
=> -> y { x[y] }
```

这让我们可以替换掉整个表达式中的同一个变量，而不会不小心改变正好有相同名字的无关变量：

```
>> expression =
     LCCall.new(
       LCCall.new(LCVariable.new(:x), LCVariable.new(:y)),
       LCFunction.new(:y, LCCall.new(LCVariable.new(:y), LCVariable.new(:x)))
     )
=> x[y][-> y { y[x] }]
>> expression.replace(:x, LCVariable.new(:z))
=> z[y][-> y { y[z] }] ❶
>> expression.replace(:y, LCVariable.new(:z))
=> x[z][-> y { y[x] }] ❷
```

❶ 在原始表达式中 x 都是自由的，所以它们都被替换掉了。

❷ 只有第一次出现的 y 才是自由变量，因此只有它被替换掉了。第二个 y 是个函数参数，不是变量，而第三个 y 是一个属于那个函数的变量，所以不应该碰它。



简单的 #replace 实现在某些输入下不能工作。它无法正确地处理含有自由变量的替换：

```
>> expression =
     LCFunction.new(:x,
       LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
     )
=> -> x { x[y] }
```

```
>> replacement = LCCall.new(LCVariable.new(:z), LCVariable.new(:x))
=> z[x]
>> expression.replace(:y, replacement)
=> -> x{ x[z[x]] }
```

像那样只是把 z[x] 粘贴进 -> x { ... } 的函数体内是不行的，因为 z[x] 中的 x 是一个自由变量，在处理完之后应该保持不变，但在这里，它恰好被同名的函数参数捕获了。[8]

我们可以忽略这个缺陷，因为我们将只对不含任何自由变量的表达式求值，因此实际上它不会产生任何问题，但是要注意，一般情况下，需要一种更为复杂的实现。

2. 调用函数

方法 #replace 的作用就是给我们一种实现函数调用语义的方式。在 Ruby 中，在用一个或者多个参数调用 proc 的时候，proc 的主体会得到求值，在这个环境下每个参数都被赋值给了一个本地变量，因此每次使用变量时都像用参数本身一样。这暗示着，用参数 1 和 2 调用 proc->x，y {x + y} 会产生中间表达式 1+2，它是为了产生最终结果所要求值的表达式。

在 lambda 演算中我们可以应用同样的思想，在对一个调用求值的时候替换一个函数体内的变量。为此，我们可以定义一个 LCFunction#call 方法，这个方法进行替换并返回结果：

```
class LCFunction
  def call(argument)
    body.replace(parameter, argument)
  end
end
```

这让我们可以模拟一个函数被调用的时刻：

```
>> function =
     LCFunction.new(:x,
       LCFunction.new(:y,
         LCCall.new(LCVariable.new(:x), LCVariable.new(:y))
       )
     )
=> -> x { -> y { x[y] } }
>> argument = LCFunction.new(:z, LCVariable.new(:z))
=> -> z { z }
>> function.call(argument)
=> -> y { -> z { z }[y] }
```

3. 规约表达式

在对一个 lambda 演算程序求值的时候，函数调用是唯一实际发生的事情，因此现在我们

注 8：正确的行为是自动改掉函数参数的名字，这样就避免与任何自由变量冲突了：->x{ x[y] } 改写为等价的表达式 ->w { w[y] }，然后再安全地执行替换，得到 ->w { w[z[x]] }，而 x 仍然是自由变量。

准备实现 #replace。它会找到表达式中函数调用能发生的地方，然后使用 #call 方法使函数调用发生。我们只需要能识别哪些表达式是实际能调用的……

```
class LCVariable
  def callable?
    false
  end
end

class LCFunction
  def callable?
    true
  end
end

class LCCall
  def callable?
    false
  end
end
```

……然后就可以写 #reduce 了：

```
class LCVariable
  def reducible?
    false
  end
end

class LCFunction
  def reducible?
    false
  end
end

class LCCall
  def reducible?
    left.reducible? || right.reducible? || left.callable?
  end

  def reduce
    if left.reducible?
      LCCall.new(left.reduce, right)
    elsif right.reducible?
      LCCall.new(left, right.reduce)
    else
      left.call(right)
    end
  end
end
```

在这个实现中，函数调用是唯一一种能被规约的语法。规约 LCCall 有点像规约 SIMPLE 里的 Add 或 Multiply：如果其中有一个子表达式可以规约，我们就对其规约；如果都不能规

约，我们就通过以右边的子表达式作为左边子表达式（应该是一个 LCFunction）的参数调用左边的子表达式来实际执行调用。这个策略称为*值调用*求值——首先我们把参数规约成一个不可规约的值，然后再执行调用。

使用 lambda 演算来计算一下"一加一"，以此来测试我们的实现：

```
>> expression = LCCall.new(LCCall.new(add, one), one)
=> -> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[-> p { -> x { p[x] }
}][-> p { -> x { p[x] } }]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
-> m { -> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][m] } }[-> p { -> x { p[x] } }]
[-> p { -> x { p[x] } }]
-> n { n[-> n { -> p { -> x { p[n[p][x]] } } }][-> p { -> x { p[x] } }] }[-> p { -> x
{ p[x] } }]
-> p { -> x { p[x] } }[-> n { -> p { -> x { p[n[p][x]] } } }][-> p { -> x { p[x] } }]
-> x { -> n { -> p { -> x { p[n[p][x]] } } }[x] }[-> p { -> x { p[x] } }]
-> n { -> p { -> x { p[n[p][x]] } } }[-> p { -> x { p[x] } }]
-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }
=> nil
```

好吧，有些事情确实发生了，不过我们没得到想要的结果：最终的表达式是 `-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }`，但数字"二"的 lambda 演算表示应该是 `-> p { -> x{ p[p[x]] } })]`。哪里错了呢？

错误是由我们使用的求值策略引起的。结果里还有可规约的函数调用——例如调用 `-> p { -> x { p[x] } }[p]` 可以被规约成 `-> x { p[x] }`——但 `#reduce` 没有接触到它们，因为它们是在一个函数体内出现的，而我们的语义不会把函数处理成可规约的。[9]

但是，就像前面 6.1.1 节中"相等"部分讨论的一样，两个具有不同语法的表达式如果有同样的行为仍然被认为是相等的。我们知道数字"二"的 lambda 演算表达式应该是：如果我们给它两个参数，它会对第二个参数调用第一个参数两次。让我们试着用两个改造过的变量 `inc` 和 `zero`[10] 调用表达式，然后看一下它实际在做什么：

```
>> inc, zero = LCVariable.new(:inc), LCVariable.new(:zero)
=> [inc, zero]
>> expression = LCCall.new(LCCall.new(expression, inc), zero)
=> -> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }[inc][zero]
>> while expression.reducible?
```

注 9：为了修正这个问题，我们可以重新实现 `#reduce` 方法，使用更激进的求值策略（如*应用序求值*或者*正则序求值*）对函数体执行规约，但处理单一函数体时通常都包含自由变量，所以需要一个 `#replace` 的更健壮的实现。

注 10：我们对含有自由变量 `inc` 和 `zero` 的表达式求值是在冒险，但幸运的是，表达式中没有一个函数含有这些名字的参数，因此在这个特例中，不管哪个变量被意外捕获都不会有危险。

```
    puts expression
    expression = expression.reduce
  end; puts expression
-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }[inc][zero]
-> x { inc[-> p { -> x { p[x] } }[inc][x]] }[zero]
inc[-> p { -> x { p[x] } }[inc][zero]]
inc[-> x { inc[x] }[zero]]
inc[inc[zero]]
=> nil
```

这恰好是我们希望数字“二”所要表现的行为，因此尽管 `-> p { -> x { p[-> p { -> x { p[x] } }[p][x]] } }` 看起来与期望的不同，但毕竟是正确的结果。

6.2.3 语法分析

既然已经有了工作语义，我们就通过为 lambda 演算表达式构建一个语法解析器来结束工作。像往常一样，我们可以使用 Treetop 来写语法：

```
grammar LambdaCalculus
  rule expression
    calls / variable / function
  end

  rule calls
    first:(variable / function) rest:('[' expression ']')+ {
      def to_ast
        arguments.map(&:to_ast).inject(first.to_ast) { |l, r| LCCall.new(l, r) }
      end

      def arguments
        rest.elements.map(&:expression)
      end
    }
  end

  rule variable
    [a-z]+ {
      def to_ast
        LCVariable.new(text_value.to_sym)
      end
    }
  end

  rule function
    '-> ' parameter:[a-z]+ ' { ' body:expression ' }' {
      def to_ast
        LCFunction.new(parameter.text_value.to_sym, body.to_ast)
      end
    }
  end
end
```



就像在 2.6 节中讨论的那样，Treetop 语法一般会产生右结合的树，因此为了适应 lambda 演算的左结合函数调用语法，这个语法得做一些额外的工作。这个调用匹配一个或者多个连续的调用（如 a[b][c][d]），而得到的具体语法树节点的 #to_ast 方法使用 Enumerable#inject 把这些调用的参数转成一个左结合的抽象语法树。

这个解析器和操作语义一起给出了 lambda 演算的完整实现，这允许我们读取表达式并对其求值：

```
>> require 'treetop'
=> true
>> Treetop.load('lambda_calculus')
=> LambdaCalculusParser
>> parse_tree = LambdaCalculusParser.new.parse('-> x { x[x] }[-> y { y }]')
=> SyntaxNode+Calls2+Calls1 offset=0, "...}[-> y { y }]" (to_ast,arguments,first,rest):
     SyntaxNode+Function1+Function0 offset=0, "... x { x[x] }" (to_ast,parameter,body):
       SyntaxNode offset=0, "-> "
       SyntaxNode offset=3, "x":
         SyntaxNode offset=3, "x"
       SyntaxNode offset=4, " { "
       SyntaxNode+Calls2+Calls1 offset=7, "x[x]" (to_ast,arguments,first,rest):
         SyntaxNode+Variable0 offset=7, "x" (to_ast):
           SyntaxNode offset=7, "x"
         SyntaxNode offset=8, "[x]":
           SyntaxNode+Calls0 offset=8, "[x]" (expression):
             SyntaxNode offset=8, "["
             SyntaxNode+Variable0 offset=9, "x" (to_ast):
               SyntaxNode offset=9, "x"
             SyntaxNode offset=10, "]"
       SyntaxNode offset=11, " }"
     SyntaxNode offset=13, "[-> y { y }]":
       SyntaxNode+Calls0 offset=13, "[-> y { y }]" (expression):
         SyntaxNode offset=13, "["
         SyntaxNode+Function1+Function0 offset=14, "... { y }" (to_ast,parameter,body):
           SyntaxNode offset=14, "-> "
           SyntaxNode offset=17, "y":
             SyntaxNode offset=17, "y"
           SyntaxNode offset=18, " { "
           SyntaxNode+Variable0 offset=21, "y" (to_ast):
             SyntaxNode offset=21, "y"
           SyntaxNode offset=22, " }"
         SyntaxNode offset=24, "]"
>> expression = parse_tree.to_ast
=> -> x { x[x] }[-> y { y }]
>> expression.reduce
=> -> y { y }[-> y { y }]
```

第7章

通用性无处不在

我们在世上见到的大多数错综复杂的事物都来自于复杂的系统，比如哺乳动物、微处理器、经济、天气，所以很自然地以为简单的系统只能做简单的事情。但在本书中，我们已经看到，简单的系统可以拥有强大的功能，例如第6章表明，即使一种很小的编程语言也有足够的能力去做有用的工作，而第5章勾勒出了一台通用图灵机的设计，它可以读取描述另一台机器的编码，然后模拟其执行。

通用图灵机的存在是极其有意义的。尽管任何一台个体的图灵机都有一个硬编码的规则手册，但是通用图灵机证明了设计这样一个装置的可能性，这个装置可以通过从纸带读取指令来完成任何任务。这些指令实际上是控制机器硬件运行的软件，就像控制我们每天都在使用的通用可编程计算机的软件一样[1]。有限和下推自动机有点过于简单，不能支持这种全面的可编程性，但是图灵机具有解决这个问题的足够的复杂性。

这一章里，我们将探寻几个简单的系统，并将看到它们都是通用的——所有这些系统都具有模拟图灵机的能力，因此都能够执行所输入的任意程序，而无需硬编码——这表明通用性比我们预期的要常见得多。

7.1 lambda演算

我们已经看到，lambda 演算是一种可用的编程语言，但还没有探讨它是否与图灵机一样强

注1：“硬件”指的是读/写头、纸带和规则手册。因为图灵机通常只是一个思维实验品而不是物理实体，所以从表面上来讲它们不是硬件，但与写在纸带上的以字符形式存在的一直在改变的“软”信息相比，它们是系统中一个固定的部分，从这个意义上讲，它们是“硬的”。

大。事实上，lambda 演算一定至少有那么强大，因为它能够模拟包括通用图灵机（当然包括）在内的任何图灵机。

我们将用 lambda 演算快速地实现一台图灵机的一部分——纸带，来领略一下它是如何模拟图灵机的。



就像在第 6 章一样，我们仍将采用 Ruby 代码来方便快捷地表示 lambda 演算，当然这些代码只限于创建 proc、调用 proc 和使用常量做缩略词。

因为 Ruby 不是我们应该研究的语言，所以使用它有点冒险。但这样做换来的是一个熟悉的表达式语法和一种对表达式求值的简单方法。并且，只要保持前面的约束，我们的发现就将是有效的。

一台图灵机的纸带有 4 个属性：出现在纸带左边的字符列表、纸带中间的字符（处于图灵机读 / 写头的位置）、右侧的字符列表，以及被当成空白的字符。我们可以把这 4 个值表示成 pair 的 pair。

```
TAPE        = -> l { -> m { -> r { -> b { PAIR[PAIR[l][m]][PAIR[r][b]] } } } }
TAPE_LEFT   = -> t { LEFT[LEFT[t]] }
TAPE_MIDDLE = -> t { RIGHT[LEFT[t]] }
TAPE_RIGHT  = -> t { LEFT[RIGHT[t]] }
TAPE_BLANK  = -> t { RIGHT[RIGHT[t]] }
```

作为“构造函数”，TAPE 用纸带的 4 个属性作为参数并返回一个代表纸带的 proc。TAPE_LEFT、TAPE_MIDDLE、TAPE_RIGHT 和 TAPE_BLANK 是“访问函数”，可以根据纸带状态的一个表示来取得对应的属性。

有了这个数据结构，我们就可以实现 TAPE_WRITE。TAPE_WRITE 把一个纸带和一个字符作为输入参数，返回一个中间位置写有字符的新纸带：

```
TAPE_WRITE = > t { -> c { TAPE[TAPE_LEFT[t]][c][TAPE_RIGHT[t]][TAPE_BLANK[t]] } }
```

我们还可以定义移动纸带头的操作。TAPE_MOVE_HEAD_RIGHT 这个 proc 直接从 5.1.4 节里 Tape#move_head_right 的无限制的 Ruby 实现转换而来，它能够把纸带头右移一个方格[2]：

```
TAPE_MOVE_HEAD_RIGHT =
  -> t {
    TAPE[
      PUSH[TAPE_LEFT[t]][TAPE_MIDDLE[t]]
    ][
      IF[IS_EMPTY[TAPE_RIGHT[t]]][
        TAPE_BLANK[t]
      ][
```

注 2：TAPE_MOVE_HEAD_LEFT 的实现类似，只是要求一些没有在 6.1.8 节中额外定义的列表操作函数。

```
        FIRST[TAPE_RIGHT[t]]
      ]
    ][
      IF[IS_EMPTY[TAPE_RIGHT[t]]][
        EMPTY
      ][
        REST[TAPE_RIGHT[t]]
      ]
    ][
      TAPE_BLANK[t]
    ]
  }
```

总而言之，这些操作给予了我们创建纸带、对纸带进行读写并来回移动纸带头所需要的一切。例如，我们可以从一个空的纸带开始，然后在连续的方格内写入一串数字。

```
>> current_tape = TAPE[EMPTY][ZERO][EMPTY][ZERO]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][ONE]
=> #<Proc (lambda)>
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][TWO]
=> #<Proc (lambda)>
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> current_tape = TAPE_WRITE[current_tape][THREE]
=> #<Proc (lambda)>
>> current_tape = TAPE_MOVE_HEAD_RIGHT[current_tape]
=> #<Proc (lambda)>
>> to_array(TAPE_LEFT[current_tape]).map { |p| to_integer(p) }
=> [1, 2, 3]
>> to_integer(TAPE_MIDDLE[current_tape])
=> 0
>> to_array(TAPE_RIGHT[current_tape]).map { |p| to_integer(p) }
=> []
```

我们将跳过其他细节，但是继续像这样基于 proc 来构建对状态、配置、规则和规则手册的表示并不困难。有了全部这些，我们就可以写出只基于 proc 的 `DTM#step` 和 `DTM#run` 的实现：`STEP` 通过对一个配置应用规则手册并生成另外一个配置，模拟了一台图灵机的一步，而 `RUN` 会使用 Z 组合子反复调用 `STEP`，直到没有规则可用或机器到达停机状态，这样就模拟了一台机器的完整执行。

换句话说，`RUN` 是一个可以模拟任何图灵机的 lambda 演算程序 [3]。事实证明，相反的情况也是可能的：就像 6.2.2 节所描述的，通过在纸带上存储一个 lambda 表达式的描述，并不断根据一系列规约规则对其进行修改，一台图灵机可以作为 lambda 演算的解释器。

注 3：术语图灵完备经常用来描述一个系统或者一种编程语言能模拟任何图灵机。



因为每一台图灵机都能由 lambda 演算程序模拟，而每一个 lambda 演算程序也能被一台图灵机模拟，所以这两个系统是完全等价的。这个结果很令人吃惊，因为图灵机和 lambda 演算程序以完全不同的方式工作，我们此前没有料到它们竟然具有同样的能力。

这意味着至少有一种方式可以模拟 lambda 演算本身：首先使用 lambda 演算实现一台图灵机，然后使用这台模拟出来的机器运行 lambda 解释器。“模拟机中再模拟”是一种低效的做事方式。我们可以通过设计数据结构表示 lambda 演算表达式，然后直接实现运算语义达到同样目的。但这确实表明 lambda 演算不必再创建任何新的东西就肯定是通用的了。自解释器是通用图灵机的 lambda 演算版本：即使底层的解释程序是固定的，我们也可以通过提供合适的 lambda 表达式作为输入来让它做任何工作。

如前所述，通用系统的真正好处是它能被编程以执行不同的任务，而不是总要硬编码来。特别地，通用系统能被编程来模拟任何其他的通用系统；通用图灵机能计算 lambda 演算表达式的值，而 lambda 演算解释器也能模拟图灵机。

7.2 部分递归函数

lambda 演算表达式完全由 procs 的创建和调用组成，*部分递归函数*与其大致相同，由四个部分组合构成。前两部分叫作 `zero` 和 `increment`，我们可以使用 Ruby 实现它们。

```
def zero
  0
end

def increment(n)
  n + 1
end
```

这两个方法很直观，分别返回数字 `0` 和往一个数字上加 `1`：

```
>> zero
=> 0
>> increment(zero)
=> 1
>> increment(increment(zero))
=> 2
```

下面使用 `#zero` 和 `#increment` 来定义一些新方法：

```
>> def two
     increment(increment(zero))
   end
=> nil
```

```
>> two
=> 2
>> def three
     increment(two)
   end
=> nil
>> three
=> 3
>> def add_three(x)
     increment(increment(increment(x)))
   end
=> nil
>> add_three(two)
=> 5
```

第三个方法 #recurse 更为复杂：

```
def recurse(f, g, *values)
  *other_values, last_value = values

  if last_value.zero?
    send(f, *other_values)
  else
    easier_last_value = last_value - 1
    easier_values = other_values + [easier_last_value]

    easier_result = recurse(f, g, *easier_values)
    send(g, *easier_values, easier_result)
  end
end
```

方法 #recurse 用两个方法的名字 f 和 g 作为参数，并且使用它们对一些输入值执行递归计算。根据最后的输入值，调用 #recurse 的直接结果是通过委托给 f 或者 g 计算得出的。

- 如果最后的输入值是零，#recurse 把其他值作为参数，调用名为 f 的方法。
- 如果最后的输入不是零，#recurse 使其递减，并用修改之后的输入值作为参数调用自身，然后用那些相同的值和递归调用的结果调用名为 g 的方法。

这听起来比实际复杂；#recurse 只不过是定义某种递归函数的模板。比如，我们可以用其定义一个函数 #add，这个函数带有两个参数 x 和 y，它把它们加到一起。为了使用 #recurse 构建此函数，我们需要实现两个其他的函数，以回答下面这些问题。

- 给定 x 的值，add(x, 0) 的值是多少？
- 给定 x、y-1 和 add(x, y-1) 的值，add(x,y) 的值是多少？

第一个问题简单：一个数字加零不会有变化，所以如果我们知道 x 的值，add(x, 0) 的值将是相同的。我们可以将其实现为一个叫 #add_zero_to_x 的函数，这个函数只返回它的参数：

```
def add_zero_to_x(x)
  x
end
```

第二个问题要难一点，但是回答起来仍然足够简单：如果已经有了 add(x, y-1) 的值，我们只要将其递增就能得到 add(x, y) 的值[4]。这意味着需要一个能增加其第三个参数值的函数（#recurse 用 x、y-1 和 add(x, y-1) 作为参数来调用它）。我们管这个函数叫 #increment_easier_result：

```
def increment_easier_result(x, easier_y, easier_result)
  increment(easier_result)
end
```

把这些放到一起我们就得到了 #add 的定义，它由 #recurse 和 #increment 构造出来：

```
def add(x, y)
  recurse(:add_zero_to_x, :increment_easier_result, x, y)
end
```



第 6 章的思路同样适用于这里：为了给表达式取方便的名字，我们只使用函数的定义，而不会偷偷地递归进它们[5]。如果想要写一个递归函数，我们需要使用 #recurse。

来确认一下 #add 在做它该做的事情：

```
>> add(two, three)
=> 5
```

看起来很好。我们可以用同样的策略来实现其他熟悉的例子，比如 #multiply...：

```
def multiply_x_by_zero(x)
  zero
end

def add_x_to_easier_result(x, easier_y, easier_result)
  add(x, easier_result)
end

def multiply(x, y)
  recurse(:multiply_x_by_zero, :add_x_to_easier_result, x, y)
end
```

注 4：因为减法是加法的逆运算，所以 (x+(y-1))+1=(x+(y+-1))+1。因为加法的结合律，所以 (x+(y+-1))+1=(x+y)+(-1+1)。而因为 -1+1=0，这在加法中是恒等式，所以 (x+y)+(-1+1)=x+y。

注 5：当然 #recurse 本身的实现从根本上使用了递归方法的定义，但这是允许的，因为我们是把 #recurse 当成系统的 4 个内建原语而不是用户定义方法来处理的。

还有 #decrement：

```
def easier_x(easier_x, easier_result)
  easier_x
end

def decrement(x)
  recurse(:zero, :easier_x, x)
end
```

还有 #subtract：

```
def subtract_zero_from_x(x)
  x
end

def decrement_easier_result(x, easier_y, easier_result)
  decrement(easier_result)
end

def subtract(x, y)
  recurse(:subtract_zero_from_x, :decrement_easier_result, x, y)
end
```

这些实现运行得都和预期一样：

```
>> multiply(two, three)
=> 6
>> def six
     multiply(two, three)
   end
=> nil
>> decrement(six)
=> 5
>> subtract(six, two)
=> 4
>> subtract(two, six)
=> 0
```

我们从 #zero、#increment 和 #recurse 组合出来的程序叫*原始递归函数*。

所有的原始递归函数都是*完全的*：不管输入什么，它们总是可以停止并返回一个结果。这是因为 #recurse 是定义递归函数的唯一合法方式，而 #recurse 是总能停止的：每一个递归调用都会使其最后一个参数更接近零，而在它最后不可避免地成为零时，递归就会停止。

方法 #zero、#increment 以及 #recurse 足以构造许多有用的函数，这其中包括图灵机执行单独一步的所有操作：一个图灵机纸带的内容可以表示成一个大数，可以用原始递归函数来读纸带头当前位置的字符、往纸带上写新的字符以及左右移动纸带头。但是，因为有些图灵机是永远循环的，所以我们没法使用原始递归函数模拟任意一台图灵机的完整运行，因此原始递归函数并不是通用的。

为了得到真正的通用系统，我们可以增加第四个基础操作——#minimize：

```
def minimize
  n = 0
  n = n + 1 until yield(n).zero?
  n
end
```

方法 #minimize 接受一个块，并不断地使用一个数字作为参数重复调用它。第一次调用时，参数是 0，然后是 1，然后是 2，之后一直用越来越大的值做参数调用块，直到返回零为止。

通过在 #zero、#increment 和 #recurse 中加入 #minimize，我们可以构造更多的函数——所有的部分递归函数——包括那些永远不会停止的函数。例如，#minimize 让我们很容易实现 #divide：

```
def divide(x, y)
  minimize { |n| subtract(increment(x), multiply(y, increment(n))) }
end
```



把表达式 subtract(increment(x), multiply(y, increment(n))) 设计成如果 y*(n+1) 大于 x 就返回零。如果试图用 13 除以 4（x=13，y=4），我们来看一下随着 n 的增长 y*(n+1) 的值的变化：

n	x	y*(n+1)	y*(n+1)比x大吗？
0	13	4	否
1	13	8	否
2	13	12	否
3	13	16	是
4	13	20	是
5	13	24	是

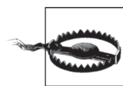
第一个满足条件的 n 值是 3，这样在 n 到达 3 的时候我们传给 #mimimize 的块会返回零，所以得到了 divide(13,4) 的结果 3。

就像原始递归函数一样，#divide 收到有意义的参数时总会返回一个结果：

```
>> divide(six, two)
=> 3
>> def ten
     increment(multiply(three, three))
   end
=> nil
>> ten
=> 10
>> divide(ten, three)
=> 3
```

但是因为 #minimize 能永远循环，所以 #divide 不一定要返回一个结果。被零除是未定义的：

```
>> divide(six, zero)
SystemStackError: stack level too deep
```



因为 #minimize 的实现是迭代的，而且没有直接增加调用栈，所以这里看到栈溢出有点奇怪，但是溢出发生在 #divide 对递归函数 #multiply 的调用期间。#multiply 的递归深度由它的第二个参数 increment(n) 决定，而随着 #minimize 的循环试图一直运行下去，n 的值变得很大，最终导致了栈溢出。

有了 #minimize，通过重复调用原始递归函数来执行模拟中的一步，就可能完全模拟一台图灵机。在停机之前模拟一直运行——如果永远不停机，那模拟就会永远运行。

7.3 SKI组合子演算

就像 lambda 演算一样，SKI 组合子演算是一个处理表达式语法的规则系统。尽管 lambda 演算已经很简单了，但仍然还有三种表达式：变量、函数和调用。我们在 6.2.2 节中看到变量使规约的规则有点复杂。SKI 演算更简单，它只有两种表达式：调用和字母符号，规则也更简单。它所有的能力都源于三个特别的符号 S、K 和 I（叫作组合子），它们每一个都有自己的归约规则：

- S[a][b][c] 规约成 a[c][b[c]]，其中 a、b 和 c 可以是任意的 SKI 演算表达式；
- K[a][b] 规约成 a；
- I[a] 规约成 a。

例如，下面是规约表达式 I[S][K][S][I[K]] 的一种方式：

```
I[S][K][S][I[K]] → S[K][S][I[K]] (规约 I[S] 为 S)
                 → S[K][S][K] (规约 I[K] 为 K)
                 → K[K][S[K]] (规约 S[K][S][K] 为 K[K][S[K]])
                 → K (reduce K[K][S[K]] 为 K)
```

注意，这里没有 lambda 演算那种变量替换，有的只是根据规约规则对符号进行的记录、复制和丢弃。

很容易实现 SKI 表达式的抽象语法：

```
class SKISymbol < Struct.new(:name)
  def to_s
    name.to_s
  end

  def inspect
    to_s
```

```
  end
end

class SKICall < Struct.new(:left, :right)
  def to_s
    "#{left}[#{right}]"
  end

  def inspect
    to_s
  end
end

class SKICombinator < SKISymbol
end

S, K, I = [:S, :K, :I].map { |name| SKICombinator.new(name) }
```



为了一般性地表示调用和符号，这里我们定义了类 SKICall 和 SKISymbol，然后创建了一次性实例 S、K 和 I 来表示作为组合子的那些特定符号。

我们没有直接让 S、K 和 I 成为 SKISymbol 的实例，而是使用了子类 SKICominator 的实例。这对我们现在没有帮助，但是它会简化以后往三个组合子对象中增加方法的工作。

这些类和对象能被用来构建 SKI 表达式的抽象语法树：

```
>> x = SKISymbol.new(:x)
=> x
>> expression = SKICall.new(SKICall.new(S, K), SKICall.new(I, x))
=> S[K][I[x]]
```

通过实现 SKI 演算的规约规则并在表达式中应用这些规则可以为 SKI 演算赋予一个小步操作语义。首先，我们将在 SKICombinator 实例上定义一个叫作 #call 的方法；S、K 和 I 都有它们自己 #call 的定义，实现了它们的归约规则：

```
# 规约 S[a][b][c] 为 a[c][b[c]]
def S.call(a, b, c)
  SKICall.new(SKICall.new(a, c), SKICall.new(b, c))
end

# 规约 K[a][b] 为 o a
def K.call(a, b)
  a
end

# 规约 I[a] 为 a
def I.call(a)
  a
end
```

好了，如果知道调用组合子的参数是什么，我们就有了一种应用演算规则的方式……

```
>> y, z = SKISymbol.new(:y), SKISymbol.new(:z)
=> [y, z]
>> S.call(x, y, z)
=> x[z][y[z]]
```

……但要对一个真正的 SKI 表达式使用 #call 方法，我们还需要从中提取出一个组合子和几个参数。因为一个表达式是用一个 SKICall 对象组成的二叉树表示的，所以这有点繁琐：

```
>> expression = SKICall.new(SKICall.new(SKICall.new(S, x), y), z)
=> S[x][y][z]
>> combinator = expression.left.left.left
=> S
>> first_argument = expression.left.left.right
=> x
>> second_argument = expression.left.right
=> y
>> third_argument = expression.right
=> z
>> combinator.call(first_argument, second_argument, third_argument)
=> x[z][y[z]]
```

为了让这个结构更容易处理，我们可以在抽象语法树上定义方法 #combinator 和 #arguments：

```
class SKISymbol
  def combinator
    self
  end

  def arguments
    []
  end
end

class SKICall
  def combinator
    left.combinator
  end

  def arguments
    left.arguments + [right]
  end
end
```

这样很容易发现要调用哪个组合子以及传给它什么参数：

```
>> expression
=> S[x][y][z]
>> combinator = expression.combinator
=> S
>> arguments = expression.arguments
=> [x, y, z]
```

```
>> combinator.call(*arguments)
=> x[z][y[z]]
```

这对 S[x][y][z] 工作得很好，但在通常情况下会有一些问题。首先 #combinator 方法只是返回一个表达式最左侧的符号，但那个符号不一定是个组合子：

```
>> expression = SKICall.new(SKICall.new(x, y), z)
=> x[y][z]
>> combinator = expression.combinator
=> x
>> arguments = expression.arguments
=> [y, z]
>> combinator.call(*arguments)
NoMethodError: undefined method `call' for x:SKISymbol
```

第二，就算最左侧的符号是一个组合子，它也不一定被用合适数目的参数调用：

```
>> expression = SKICall.new(SKICall.new(S, x), y)
=> S[x][y]
>> combinator = expression.combinator
=> S
>> arguments = expression.arguments
=> [x, y]
>> combinator.call(*arguments)
ArgumentError: wrong number of arguments (2 for 3)
```

为了避免这两个问题，我们将定义 #callable? 方法以检测是否适合以方法 #combinator 和 #argument 的结果来使用 #call。一个符号永远都无法调用，而一个组合子只有在参数个数正确的情况下才可以调用：

```
class SKISymbol
  def callable?(*arguments)
    false
  end
end

def S.callable?(*arguments)
  arguments.length == 3
end

def K.callable?(*arguments)
  arguments.length == 2
end

def I.callable?(*arguments)
  arguments.length == 1
end
```



顺便说一下，Ruby 已经有办法回答一个方法需要多少个参数了（它的参数数量）：

```
>> def add(x, y)
     x + y
   end
=> nil
>> add_method = method(:add)
=> #<Method: Object#add>
>> add_method.arity
=> 2
```

因此，我们可以用一个共享 #callable 实现来替换 S、K 和 I 各自的实现：

```
class SKICombinator
  def callable?(*arguments)
    arguments.length == method(:call).arity
  end
end
```

现在可以识别归约规则直接适用的表达式了：

```
>> expression = SKICall.new(SKICall.new(x, y), z)
=> x[y][z]
>> expression.combinator.callable?(*expression.arguments)
=> false
>> expression = SKICall.new(SKICall.new(S, x), y)
=> S[x][y]
>> expression.combinator.callable?(*expression.arguments)
=> false
>> expression = SKICall.new(SKICall.new(SKICall.new(S, x), y), z)
=> S[x][y][z]
>> expression.combinator.callable?(*expression.arguments)
=> true
```

最后，我们可以为 SKI 表达式实现熟悉的 #reducible? 和 #reduce 方法了：

```
class SKISymbol
  def reducible?
    false
  end
end

class SKICall
  def reducible?
    left.reducible? || right.reducible? || combinator.callable?(*arguments)
  end

  def reduce
    if left.reducible?
      SKICall.new(left.reduce, right)
    elsif right.reducible?
      SKICall.new(left, right.reduce)
    else
```

```
      combinator.call(*arguments)
    end
  end
end
```



SKICall#reduce 递归查找我们已经知道如何规约的子表达式（例如正在以三个参数进行调用的 S 组合子），然后使用 #call 应用合适的规则。

那就是它了！我们现在可以对 SKI 表达式不断规约，直到不能规约为止。例如，下面使用符号 x 和 y 调用表达式 S[K[S[I]]][K]，它交换了两个参数的顺序：

```
>> swap = SKICall.new(SKICall.new(S, SKICall.new(K, SKICall.new(S, I))), K)
=> S[K[S[I]]][K]
>> expression = SKICall.new(SKICall.new(swap, x), y)
=> S[K[S[I]]][K][x][y]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[K[S[I]]][K][x][y]
K[S[I]][x][K[x]][y]
S[I][K[x]][y]
I[y][K[x][y]]
y[K[x][y]]
y[x]
=> nil
```

SKI 演算用三个简单的规则就产生了出人意料的复杂行为。事实上，复杂到被证明是通用的了。我们可以证明 SKI 表达式的通用性，方法是展示如何把任意的 lambda 演算表达式转换成做同样事情的一个 SKI 表达式，这实际上也是使用 SKI 演算给了 lambda 演算一个指称语义。我们已经知道 lambda 演算是通用的，因此如果 SKI 能完全模拟它，就能得出 SKI 演算也是通用的结论。

转换的核心是一个叫 #as_a_function_of 的方法：

```
class SKISymbol
  def as_a_function_of(name)
    if self.name == name
      I
    else
      SKICall.new(K, self)
    end
  end
end

class SKICombinator
  def as_a_function_of(name)
    SKICall.new(K, self)
  end
```

```
end

class SKICall
  def as_a_function_of(name)
    left_function = left.as_a_function_of(name)
    right_function = right.as_a_function_of(name)

    SKICall.new(SKICall.new(S, left_function), right_function)
  end
end
```

方法 #as_a_function_of 的工作细节并不重要，但粗略上讲，它把一个 SKI 表达式转成一个新的表达式，这个表达式在用一个参数调用时会转回到原来的表达式。例如，表达式 S[K][I] 被转成 S[S[K[S]][K[K]]][K[I]]：

```
>> original = SKICall.new(SKICall.new(S, K), I)
=> S[K][I]
>> function = original.as_a_function_of(:x)
=> S[S[K[S]][K[K]]][K[I]]
>> function.reducible?
=> false
```

在 S[S[K[S]][K[K]]][K[I]] 以一个参数比如说 y 进行调用的时候，它将会规约回 S[K][I]：

```
>> expression = SKICall.new(function, y)
=> S[S[K[S]][K[K]]][K[I]][y]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[S[K[S]][K[K]]][K[I]][y]
S[K[S]][K[K]][y][K[I][y]]
K[S][y][K[K][y]][K[I][y]]
S[K[K][y]][K[I][y]]
S[K][K[I][y]]
S[K][I]
=> nil
>> expression == original
=> true
```

只是在原始表达式也包含有那个名字的符号时参数 name 才会用到。在那种情况下，#as_a_function_of 会产生一些更有意思的东西：一个表达式，在使用一个参数进行调用的时候，它会规约成原始表达式，其中那个参数会替换掉符号：

```
>> original = SKICall.new(SKICall.new(S, x), I)
=> S[x][I]
>> function = original.as_a_function_of(:x)
=> S[S[K[S]][I]][K[I]]
>> expression = SKICall.new(function, y)
=> S[S[K[S]][I]][K[I]][y]
>> while expression.reducible?
     puts expression
```

```
    expression = expression.reduce
  end; puts expression
S[S[K[S]][I]][K[I]][y]
S[K[S]][I][y][K[I][y]]
K[S][y][I[y]][K[I][y]]
S[I[y]][K[I][y]]
S[y][K[I][y]]
S[y][I]
=> nil
>> expression == original
=> false
```

一个 lambda 演算函数在被调用时，函数体内的变量会被替换掉，上面是对这种方式的一个明确的重新实现。本质上说，#as_a_function_of 给了我们使用 SKI 表达式作为函数体的方法：它创建了一个新的表达式，这个表达式的行为就像带有一个特定函数体和一个参数名的函数，只不过 SKI 演算没有函数语法而已。

SKI 演算模拟函数的能力把 lambda 演算表达式与 SKI 表达式的转换变得直接。lambda 演算变量和调用成为了 SKI 演算的符号和调用，而每一个 lambda 演算函数体用 #as_a_function_of 转成了一个 SKI 演算“函数”：

```
class LCVariable
  def to_ski
    SKISymbol.new(name)
  end
end

class LCCall
  def to_ski
    SKICall.new(left.to_ski, right.to_ski)
  end
end

class LCFunction
  def to_ski
    body.to_ski.as_a_function_of(parameter)
  end
end
```

让我们通过把数字“2”（参见 6.1.3 节）的 lambda 演算表示转成 SKI 演算来检查一下这个转换：

```
>> two = LambdaCalculusParser.new.parse('-> p { -> x { p[p[x]] } }').to_ast
=> -> p { -> x { p[p[x]] } }
>> two.to_ski
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]
```

SKI 演算表达式 S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]] 与 ->p{->x{p[p[x]]}} 做的事情一样吗？应该是在其第二个参数上调用它的第一个参数两次，因此我们可以尝试给它一些参数来看看它实际是怎么做的，就像在 6.2.2 节看到的那样：

```
>> inc, zero = SKISymbol.new(:inc), SKISymbol.new(:zero)
=> [inc, zero]
>> expression = SKICall.new(SKICall.new(two.to_ski, inc), zero)
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]][inc][zero]
S[K[S]][S[K[K]][I]][inc][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
K[S][inc][S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[S[K[K]][I][inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[K][inc][I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[I[inc]]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[S[K[S]][S[K[K]][I]]][K[I]][inc]][zero]
S[K[inc]][S[K[S]][S[K[K]][I]][inc][K[I][inc]]][zero]
S[K[inc]][K[S][inc][S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[S[K[K]][I][inc]][K[I][inc]]][zero]
S[K[inc]][S[K[K][inc][I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[I[inc]]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][K[I][inc]]][zero]
S[K[inc]][S[K[inc]][I]][zero]
K[inc][zero][S[K[inc]][I][zero]]
inc[S[K[inc]][I][zero]]
inc[K[inc][zero][I[zero]]]
inc[inc[I[zero]]]
inc[inc[zero]]
=> nil
```

可以确定了，使用叫 inc 和 zero 的符号调用转换过的表达式求值为 inc[inc[zero]]，这正是我们所想要的。同样的转换对任何其他 lambda 表达式也能成功执行，因此 SKI 组合子演算可以完全模拟 lambda 演算，从而它一定是通用的。



尽管 SKI 演算有三个组合子，但 I 组合子实际上是冗余的。有许多表达式只含有 S 和 K，它们做的事情和 I 一样；例如 S[K][K]：

```
>> identity = SKICall.new(SKICall.new(S, K), K)
=> S[K][K]
>> expression = SKICall.new(identity, x)
=> S[K][K][x]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[K][K][x]
K[x][K[x]]
x
=> nil
```

可见 S[K][K] 的行为与 I 一样，这对任何形式为 S[K][任意] 的 SKI 表达式都成立。I 组合子是我们非必需的语法糖。对于通用性来说，两个组合子 S 和 K 就足够了。

7.4　约塔（Iota）

希腊字母约塔（ι）是可以添加到 SKI 演算里的另一个组合子。下面是它的规约规则：ι[α] 可以规约成 α[S][K]。

我们的 SKI 演算实现让加入一个新的组合子变得很容易：

```
IOTA = SKICombinator.new('ι')

# 规约 ι[a] 为 a[S][K]
def IOTA.call(a)
  SKICall.new(SKICall.new(a, S), K)
end

def IOTA.callable?(*arguments)
  arguments.length == 1
end
```

Chris Barker 提交了一种叫作 Iota（http://semarch.linguistics.fas.nyu.edu/barker/Iota/）的语言，它的程序只使用 ι 组合子。尽管只有一个组合子，Iota 仍然是一种通用语言，因为任何 SKI 演算表达式都可以转成它，而我们已经看到 SKI 演算是通用的。

可以通过应用这些替换规则把 SKI 表达式转成 Iota：

- 用 ι[ι[ι[ι[ι]]]] 替换 S；
- 用 ι[ι[ι[ι]]] 替换 K；
- 用 ι[ι] 替换 I。

很容易实现这个转换：

```
class SKISymbol
  def to_iota
    self
  end
end

class SKICall
  def to_iota
    SKICall.new(left.to_iota, right.to_iota)
  end
end

def S.to_iota
  SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, IOTA))))
end

def K.to_iota
  SKICall.new(IOTA, SKICall.new(IOTA, SKICall.new(IOTA, IOTA)))
end
```

```
def I.to_iota
  SKICall.new(IOTA, IOTA)
end
```

S、K 和 I 组合子的 Iota 版与原始表达式是否等价一点都不明显，因此我们可以通过规约 SKI 演算内部的每一个组合子并观察它们的行为来进行研究。下面是在我们把 S 转换成 Iota 然后对其进行规约的过程：

```
>> expression = S.to_iota
=> ι[ι[ι[ι[ι]]]]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
ι[ι[ι[ι[ι]]]]
ι[ι[ι[ι[S][K]]]]
ι[ι[ι[S[S][K][K]]]]
ι[ι[ι[S[K][K[K]]]]]
ι[ι[S[K][K[K]][S][K]]]
ι[ι[K[S][K[K][S]][K]]]
ι[ι[K[S][K][K]]]
ι[ι[S[K]]]
ι[S[K][S][K]]
ι[K[K][S[K]]]
ι[K]
K[S][K]
S
=> nil
```

是的，ι[ι[ι[ι[ι]]]] 实际上与 S 等价。这同样也适用于 K：

```
>> expression = K.to_iota
=> ι[ι[ι[ι]]]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
ι[ι[ι[ι]]]
ι[ι[ι[S][K]]]
ι[ι[S[S][K][K]]]
ι[ι[S[K][K[K]]]]
ι[S[K][K[K]][S][K]]
ι[K[S][K[K][S]][K]]
ι[K[S][K][K]]
ι[S[K]]
S[K][S][K]
K[K][S[K]]
K
=> nil
```

但对于 I 则不行。ι 规约规则只会产生含有 S 和 K 组合子的表达式，因此不可能以字面量 I 结束：

```
>> expression = I.to_iota
=> ι[ι]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
ι[ι]
ι[S][K]
S[S][K][K]
S[K][K[K]]
=> nil
```

因此 S[K][K[K]] 在语法上与 I 不等价，但它是 S 和 K 组合子表达式与 I 表达式做同样事情的另一个例子：

```
>> identity = SKICall.new(SKICall.new(S, K), SKICall.new(K, K))
=> S[K][K[K]]
>> expression = SKICall.new(identity, x)
=> S[K][K[K]][x]
>> while expression.reducible?
     puts expression
     expression = expression.reduce
   end; puts expression
S[K][K[K]][x]
K[x][K[K][x]]
K[x][K]
x
=> nil
```

所以到 Iota 的转换虽然没有完全保留所有三个 SKI 组合子的语法，但确实保留了它们的个体行为。我们可以通过把熟悉的 lambda 演算表达式用它的 SKI 演算表示转成 Iota 来测试整体的效果，然后对其求值以检查它的行为：

```
>> two
=> -> p { -> x { p[p[x]] } }
>> two.to_ski
=> S[S[K[S]][S[K[K]][I]]][S[S[K[S]][S[K[K]][I]]][K[I]]]
>> two.to_ski.to_iota
=> ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[
ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι]]
]][ι[ι[ι[ι]]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]]][ι[ι]]]]
>> expression = SKICall.new(SKICall.new(two.to_ski.to_iota, inc), zero)
=> ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[
ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι]]
]][ι[ι[ι[ι]]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]]][ι[ι]]]][inc][zero]
>> expression = expression.reduce while expression.reducible?
=> nil
>> expression
=> inc[inc[zero]]
```

inc[inc[zero]] 是我们所期望的结果，因此 Iota 表达式 ι[ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι]]]]][ι[ι]]]]

[ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]]][ι[ι[ι[ι[ι]]]]]][ι[ι[ι[ι[ι]]]]][ι[ι[ι[ι]]][ι[ι[ι[ι]]]]][ι[ι]]]][ι[ι[ι[ι]]][ι[ι]]]] 实际是一个对 ->p{->x{p[p[x]]}} 进行无变量、无函数并且只有一个组合子的有效转换。而因为我们可以对任何 lambda 演算表达式进行这种转换，所以 Iota 是另一种通用语言。

7.5 标签系统

标签系统（tag system）是一个类似简化版图灵机的计算模型：标签系统不是在一条纸带上来回移动纸带头，而是反复在一个字符串的末尾增加新的字符并在开头处移除字符。在某方面，标签系统的字符串像是图灵机的纸带，但标签系统被限定在只能在字符串的两头操作，而且它只能朝着末尾“移动”。

标签系统的描述包括两部分：首先，一个规则集合，其中每一条规则定义当特定的字符出现在字符串的开头时，要给这个字符串添加的一些字符（例如“字符串的开头是字符 a 时，添加字符 bcd”）；其次，一个叫作*删除数*的数字，它定义了按照一个规则执行之后有多少字符要从字符串的开头删除。

下面是一个标签系统的例子：

- 字符串以 a 开头时，添加字符 bc；
- 字符串以 b 开头时，添加字符 caad；
- 字符串以 c 开头时，添加字符 ccd；
- 按照上面的任何规则执行之后，从字符串的开头删除三个字符，换句话说，删除数是 3。

我们可以通过反复遵照规则并删除字符直到字符串的首字符没有可用的规则，或者直到字符串的长度小于删除数[6]，以此来执行一个标签系统的计算。我们用初始字符串 'aaaaaa' 来运行一下示例标签系统：

当前字符串	可用规则
aaaaaa	字符串以 a 开头时，添加字符 bc
aaa**bc**	字符串以 a 开头时，添加字符 bc
bc**bc**	字符串以 b 开头时，添加字符 caad
c**caad**	字符串以 c 开头时，添加字符 ccd
ad**ccd**	字符串以 a 开头时，添加字符 bc
cd**bc**	字符串以 c 开头时，添加字符 ccd
cccd	字符串以 c 开头时，添加字符 ccd
d**ccd**	—

注 6：第二个条件可以防止我们删除比字符串所含有的字符数还多的字符。

标签系统只能直接在字符串上操作，但我们也可以让它们对其他类型的值（例如数字）执行复杂的操作，只要用合适的方式把那些值编码成字符串就行。对数字编码的一种可能方式是：把数字 n 表示成字符串 aa 后跟重复 n 次的字符串 bb。例如，把数字 3 表示成字符串 aabbbbbb。



这个表示的某些方面可能看起来是多余的（可以只是把 3 表示成 aaa），但很快你就会发现，使用成对的字符，并在字符串的开头进行明确的标记很有用。

选定了数字的编码模式，就可以设计标签系统操作数字了。下面是一个对输入数翻倍的系统：

- 字符串以 a 开头时，添加字符 aa；
- 字符串以 b 开头时，添加字符 bbbb；
- 在执行完一个规则之后，从字符串的开头删掉两个字符（删除数为 2）。

观察一下起始字符串是 aabbbb 时这个标签系统是如何表现的，这个字符串表示 2：

```
aabbbb → bbbbaa
       → bbaabbbb
       → aabbbbbbbb（表示数字 4）
       → bbbbbbbbaa
       → bbbbbbaabbbb
       → bbbbaabbbbbbbb
       → bbaabbbbbbbbbbbb
       → aabbbbbbbbbbbbbbbb（数字 8）
       → bbbbbbbbbbbbbbbbaa
       → bbbbbbbbbbbbbbaabbbb
       ...
```

很明显翻倍了，但这个标签系统却永远运行下去了（把由当前字符串表示的数翻倍，然后再翻倍，然后再翻倍），这真不是我们想要的。为了设计一个只对一个数字翻倍一次然后停机的系统，我们需要使用不同的字符对结果进行编码，以保证不再触发新一轮的翻倍。我们可以通过放松编码模式，允许字符 c 和 d 替换 a 和 b，然后修改规则，在表示翻倍之后的数时使用 cc 和 dddd 而不是 aa 和 bbbb。

这样改变之后，计算看起来像是这样：

```
aabbbb → bbbbcc
       → bbccdddd
       → ccdddddddd（数字 4，用 c 和 d 而不是 a 和 b 进行编码）
```

修改后的系统在到达 ccdddddddd 时会停止，因为没有针对 c 开头字符串的规则。



在这种情况下，我们只是依赖字符 c 适时停止计算，因此完全可以在结果中重用 b 而不是用 d 来替换它，但使用超出必要的字符没有什么害处。

使用不同的字符集合来对输入和输出值进行编码会更清晰一些。就像我们很快将要看到的那样，这还能更容易地把几个小的标签系统组合成一个大的系统，可以通过把一个系统的输出编码与另一个系统的输入编码匹配做到。

为了在 Ruby 中模拟标签系统，我们需要一个单规则的实现（TagRule），一个规则集合的实现（TagRulebook），以及标签系统自身的实现（TagSystem）：

```
class TagRule < Struct.new(:first_character, :append_characters)
  def applies_to?(string)
    string.chars.first == first_character
  end

  def follow(string)
    string + append_characters
  end
end

class TagRulebook < Struct.new(:deletion_number, :rules)
  def next_string(string)
    rule_for(string).follow(string).slice(deletion_number..-1)
  end

  def rule_for(string)
    rules.detect { |r| r.applies_to?(string) }
  end
end

class TagSystem < Struct.new(:current_string, :rulebook)
  def step
    self.current_string = rulebook.next_string(current_string)
  end
end
```

这个实现允许我们单步执行标签系统的计算，一次只执行一个规则。让我们试试之前的对数字翻倍的例子，这次对数字 3（aabbbbbb）翻倍：

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'aa'), TagRule.new('b', 'bbbb')])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbbbb', rulebook)
=> #<struct TagSystem ...>
>> 4.times do
     puts system.current_string
     system.step
   end; puts system.current_string
aabbbbbb
bbbbbbaa
bbbbaabbbb
```

```
bbaabbbbbbbb
aabbbbbbbbbbbb
=> nil
```

因为这个标签系统会永远运行，所以我们只能在结果出现之前预先知道执行多少步（这种情况下是 4 步），但如果我们使用把结果用 c 和 d 编码的修改版本，就可以让它自动停下来。增加代码来支持它：

```
class TagRulebook
  def applies_to?(string)
    !rule_for(string).nil? && string.length >= deletion_number
  end
end

class TagSystem
  def run
    while rulebook.applies_to?(current_string)
      puts current_string
      step
    end

    puts current_string
  end
end
```

现在可以只对标签系统的停机版本调用 TagSystem#run，并让其在合适时机自然停止：

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'cc'), TagRule.new('b', 'dddd')])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbbbb
bbbbbbcc
bbbbccdddd
bbccdddddddd
ccdddddddddddd
=> nil
```

这个实现允许我们探索标签系统能做的其他事情。使用我们的编码模式，很容易设计系统执行其他的数字操作，就像下面这个对一个数字减半的系统：

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'cc'), TagRule.new('b', 'd')])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbbbbbbbbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbbbbbbbbbb
bbbbbbbbbbbbcc
bbbbbbbbbbccd
bbbbbbbbccdd
bbbbbbccddd
```

```
bbbbccdddd
bbccddddd
ccdddddd
=> nil
```

还有这个递增一个数字的系统：

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'ccdd'), TagRule.new('b', 'dd')])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbb
bbbbccdd
bbccdddd
ccdddddd
=> nil
```

我们可以把两个标签系统联结一起，只要第一个系统的输出编码与第二个系统的输入编码匹配即可。下面是一个简单的系统，它使用字符 c 和 d 对递增规则的输入进行编码，并用 e 和 f 对它们的输出进行编码，以此把翻倍和递增规则组合到一起：

```
>> rulebook = TagRulebook.new(2, [
     TagRule.new('a', 'cc'), TagRule.new('b', 'dddd'), # double
     TagRule.new('c', 'eeff'), TagRule.new('d', 'ff')  # increment
   ])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbb（数字 2）
bbbbcc
bbccdddd
ccdddddddd（数字 4） ❶
ddddddddeeff
ddddddeeffff
ddddeeffffff
ddeeffffffff
eeffffffffff（数字 5）❷
=> nil
```

❶ 翻倍规则把 2 转成 4，用字符 c 和 d 编码。

❷ 递增规则把 4 转成 5，这次使用 e 和 f 编码。

除了把数字转成其他数字之外，标签系统还可以检查它们的数学特性。下面是测试一个数是奇数还是偶数的标签系统：

```
>> rulebook = TagRulebook.new(2, [
     TagRule.new('a', 'cc'), TagRule.new('b', 'd'),
     TagRule.new('c', 'eo'), TagRule.new('d', ''),
     TagRule.new('e', 'e')
   ])
=> #<struct TagRulebook ...>
```

如果输入代表一个偶数，这个系统会停止在单字符 e（代表“偶数”）：

```
>> system = TagSystem.new('aabbbbbbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbbbbbb (the number 4)
bbbbbbbbcc
bbbbbbccd
bbbbccdd
bbccddd
ccdddd   ❶
ddddeo   ❷
ddeo
eo ❸
e  ❹
=> nil
```

❶ a 和 b 把输入减半；ccdddd 代表数字 2。

❷ c 规则删掉前导的 cc 对，并添加字符 eo，它们中间的一个会形成最后的结果。

❸ 空的 d 规则会耗尽所有的前导 dd 对，只留下 eo。

❹ e 规则只会用 e 替换 eo，然后系统停机。

如果输入的数为奇数，那么结果就是字符串 o（代表“奇数”）：

```
>> system = TagSystem.new('aabbbbbbbbbb', rulebook)
=> #<struct TagSystem ...>
>> system.run
aabbbbbbbbbb (数字 5)
bbbbbbbbbbcc
bbbbbbbbccd
bbbbbbccdd
bbbbccddd
bbccdddd
ccddddd ❶
dddddeo
dddeo
deo ❷
o ❸
=> nil
```

❶ 数字像以前一样减半，但因为这次是奇数，所以结果是一个奇数个 d 组成的字符串。我们对数字的编码模式只使用成对的字符，因此 ccddddd 不代表任何数，但因为它含有“两个半”成对的字符 d，可以不正式地把它看成是数字 2.5。

❷ 所有前导的 dd 对都被删掉了，在最终的 eo 之前留下了一个 d。

❸ 残留的 d 被删掉了，并带走了 e，只留下 o，然后系统停机。



为了让这个标签系统工作，拥有大于 1 的删除数至关重要。因为每个第二字符都会触发一个规则，我们可以通过在特定的触发位置安排特定的字符出现（或者不出现）来影响系统的行为。这种让字符在删除行为中同步或者不同步出现的技术是设计强大标签系统的关键。

这些数字操作技术可以用来模拟一台图灵机。在像标签系统这么简单的东西之上构建模拟的图灵机涉及大量细节，但其中一种工作方式像是下面这样。

(1) 作为可能最简单的例子，让一台图灵机的纸带只使用两个字符，我们将称它们为 0 和 1，其中 0 扮演空白字符的角色。
(2) 把图灵机的纸带分成两部分：左半部分含有纸带头下的字符和所有它左边的字符，右半部分含有纸带头右边的所有字符。
(3) 把纸带的左半部分作为一个二进制数：如果最初的纸带类似 0001101(0)0011000，那么左半部分就是二进制数 11010，这是十进制数 26。
(4) 把纸带的右半部分作为一个*反写*的二进制数：示例纸带的右半部分是二进制数 1100，即十进制数 12。
(5) 把这两个数编码成一个适合由标签系统使用的字符串。对于示例纸带，我们可以使用 aa 后跟 26 份 bb，然后 cc 后跟 12 份 dd。
(6) 使用简单的翻倍、减半、递增、递减，以及奇偶检查模拟从纸带上读、向纸带写以及移动纸带头。例如，我们通过对左半部分数字翻倍，对右半部分数字减半[7]来在示例纸带上向右移动纸带头：翻倍 26 得到 52，二进制就是 110100；12 的一半是 6，二进制是 110。因此新的纸带看起来是 011010(0)011000。从纸带上读取意味着检查表示纸带左半部分的数字是奇数还是偶数，而向纸带上写一个 1 或者 0 意思是对那个数递增或者递减。
(7) 使用选择的字符来对左右纸带数进行编码，以此来表示所模拟图灵机的当前状态：或许机器处于状态 1，我们使用 a、b、c 和 d 来对纸带进行编码，但它转移到状态 2 时，使用 e、f、g 和 h 来编码，以此类推。
(8) 把每一个图灵机规则转成一个标签系统，它会用合适的方式对当前字符串进行重写。读取一个 0，写入一个 1，向右移动纸带头并进入状态 2 的规则变成的标签系统，会检查左侧纸带的数是偶数，对其递增，翻倍左边纸带的数，同时减半右边纸带的数，然后产生一个使用状态 2 的字符编码的字符串。
(9) 把这些独立的标签系统组合起来，就是一个可以模拟图灵机每一条规则的大系统。



对于标签系统如何模拟图灵机工作的完整说明，请看 Matthew Cook 在 http://www.complex-systems.com/pdf/15-1-1.pdf 中 2.1 节所做的简洁解释。

Cook 的模拟比这里描述的更复杂。它使用当前字符串的“对齐”来表示所模拟纸带头下面的字符，而不是把它作为纸带的一部分，而且它很容易扩展，通过增加标签系统的删除数来以任意数目的字符模拟一台图灵机。

标签系统可以模拟任意图灵机的事实，意味着它也是通用的。

注 7：对一个数翻倍在二进制表示上就是所有的数字左移一位，而减半就是把所有的数字右移一位。

7.6 循环标签系统

循环标签系统（cyclic tag system）是施加了一些额外限制的更简单的标签系统。

- 循环标签系统的字符串只能包含两个字符：0 和 1。
- 循环标签系统的规则只会在当前字符串以 1 开始而不是 0 开始的时候才会应用。[8]
- 循环标签系统的删除数总是 1。

这些约束本身对于支持任何有用的计算来说都过于苛刻了，因此作为补偿循环标签系统有一个额外的特性：循环标签系统的规则手册中的第一条规则是执行开始时的当前规则，并且在计算的每一步之后，规则手册中的下一个规则就成为了当前规则，在到达规则手册结尾的时候又会回到第一个规则。

这种系统被称为“循环的”，是因为当前规则不断地在规则手册中循环。一个当前规则，再结合上每条规则都只会应用到 1 开头的字符串这一约束，避免了在每一步执行中不得不遍历规则手册查找可用规则的开销。如果首字符是 1，那么就应用当前规则，否则，就没有可用的规则。

作为一个例子，我们看一下拥有三个规则的循环标签系统，三个规则分别添加字符 1，0010 和 10。下面是以字符串 11 开始时的情形：

当前字符串	当前规则	可以应用规则吗
11	添加字符 1	是
1**1**	添加字符 0010	是
1**0010**	添加字符 10	是
0010**10**	添加字符 1	否
01010	添加字符 0010	否
1010	添加字符 10	是
010**10**	添加字符 1	否
1010	添加字符 0010	是
010**0010**	添加字符 10	否
100010	添加字符 1	是
00010**1**	添加字符 0010	否
00101	添加字符 10	否
0101	添加字符 1	否
101	添加字符 0010	是
01**0010**	添加字符 10	否
10010	添加字符 1	是
0010**1**	添加字符 0010	否
⋮	⋮	⋮

注 8：循环标签系统的规则没有必要说“字符串以 1 开始时，添加字符 011”，因为第一部分已经假定了——只需要“添加字符 011”就足够了。

尽管这个系统极其简单，我们也能看到一点点复杂的行为：接下来要发生什么并不明显。稍微思考一下，可以证明这个系统将会永远运行下去而不是缩减成一个空字符串，这是因为每个规则都添加一个 1，因此只要最初的字符串含有一个 1，它就不会完全结束。[9] 但是当前字符串会断断续续地持续变长，还是会进入扩张和收缩的反复模式呢？只看规则没法回答这个问题，需要一直运行这个系统以查明会发生什么。

我们已经有了常见标签系统的一个 Ruby 实现，因此模拟循环标签系统不需要太多的额外工作。我们通过简单的子类化 TagRule 实现 CyclicTagRule 并把 '1' 硬编码为它的 first_character：

```
class CyclicTagRule < TagRule
  FIRST_CHARACTER = '1'

  def initialize(append_characters)
    super(FIRST_CHARACTER, append_characters)
  end

  def inspect
    "#<CyclicTagRule #{append_characters.inspect}>"
  end
end
```



#initialize 是一个构造方法，在一个类的实例被创建时会自动调用。CyclicTagRule#initialize 从超类 TagRule 调用构造函数，以此来设置 first_character 和 append_character 属性。

循环标签系统的规则工作方式有些许的不同，因此我们将从头构建一个 CylicTagRulebook 类，提供对 #applies_to? 和 #next_string 的新实现：

```
class CyclicTagRulebook < Struct.new(:rules)
  DELETION_NUMBER = 1

  def initialize(rules)
    super(rules.cycle)
  end

  def applies_to?(string)
    string.length >= DELETION_NUMBER
  end

  def next_string(string)
    follow_next_rule(string).slice(DELETION_NUMBER..-1)
  end
```

注 9：循环标签系统与正常的标签系统不同，它在没有规则可用的时候仍然会一直运行，不然的话它就什么也做不了。让循环标签系统停止运行的唯一方式就是使它的当前字符串成为空。例如在初始字符串完全由字符 0 组成的时候，总会出现空字符串。

```
  def follow_next_rule(string)
    rule = rules.next

    if rule.applies_to?(string)
      rule.follow(string)
    else
      string
    end
  end
end
```

不像 TagRulebook，即使当前规则不能应用，CyclicTagRulebook 也总是应用到非空字符串上。



Array#cycle 创建一个 Enumerator（参见 6.1.11 节“原始 Ruby 流”部分），它会永远地循环访问一个数组的元素：

```
>> numbers = [1, 2, 3].cycle
=> #<Enumerator: [1, 2, 3]:cycle>
>> numbers.next
=> 1
>> numbers.next
=> 2
>> numbers.next
=> 3
>> numbers.next
=> 1
>> [:a, :b, :c, :d].cycle.take(10)
=> [:a, :b, :c, :d, :a, :b, :c, :d, :a, :b]
```

这恰好是我们对循环标签系统当前规则所要求的行为，因此 CyclicTagRulebook#initialize 把这些循环中的一个赋给规则属性，然后每次对 #follow_next_rule 的调用都使用 rules.next 得到循环中的下一条规则。

现在我们可以创建由 CyclicTagRules 组成的 CyclicTagRulebook，然后把它放到一个 TagSystem 里观察其工作情况：

```
>> rulebook = CyclicTagRulebook.new([
     CyclicTagRule.new('1'), CyclicTagRule.new('0010'), CyclicTagRule.new('10')
   ])
=> #<struct CyclicTagRulebook ...>
>> system = TagSystem.new('11', rulebook)
=> #<struct TagSystem ...>
>> 16.times do
     puts system.current_string
     system.step
   end; puts system.current_string
11
11
10010
001010
```

```
01010
1010
01010
1010
0100010
100010
000101
00101
0101
101
010010
10010
00101
=> nil
```

这与我们手工单步执行时候看到的行为相同。继续吧：

```
>> 20.times do
     puts system.current_string
     system.step
   end; puts system.current_string
00101
0101
101
011
11
110
101
010010
10010
00101
0101
101
011
11
110
101
010010
10010
00101
0101
101
=> nil
```

以字符串 11 开始时，这个系统确实进入到重复的行为中：在一段不稳定阶段过后，会出现 9 个连续的字符串（101、010010、10010、00101……）并会一直这么重复下去。当然，如果我们改变了初始字符串或者任意规则，那长期的行为都会变得不同。

循环标签系统极其受限（它们的规则不灵活，只有两个字符，删除数也是最低值），但令人吃惊的是，仍然可以使用它们模拟任何标签系统。

由一个循环标签系统对一个正常标签系统的模拟大概像下面描述的这样工作。

(1) 决定标签系统的*字母表*：它使用的字符集合。
(2) 设计编码模式，把每一个字符与一个适合用在循环标签系统里的唯一字符串关联起来（也就是只包含 0 和 1）。
(3) 把每一个原始系统的规则转换成一个循环标签系统的规则，方法是对它添加的字符进行编码。
(4) 用空规则填补循环标签系统的规则手册，模拟原始标签系统的删除数。
(5) 对原始标签系统的输入字符串进行编码，并使用它作为循环标签系统的输入。

下面就来具体实现上述思路。首先，需要能得到一个标签系统所使用的字符：

```
class TagRule
  def alphabet
    ([first_character] + append_characters.chars.entries).uniq
  end
end

class TagRulebook
  def alphabet
    rules.flat_map(&:alphabet).uniq
  end
end

class TagSystem
  def alphabet
    (rulebook.alphabet + current_string.chars.entries).uniq.sort
  end
end
```

我们可以在 7.5 节数字递增的标签系统上测试这个功能。TagSystem#alphabet 表明这个系统使用字符 a、b、c 和 d：

```
>> rulebook = TagRulebook.new(2, [TagRule.new('a', 'ccdd'), TagRule.new('b', 'dd')])
=> #<struct TagRulebook ...>
>> system = TagSystem.new('aabbbb', rulebook)
=> #<struct TagSystem ...>
>> system.alphabet
=> ["a", "b", "c", "d"]
```

下一步，我们需要把每个字符编码成循环标签系统能使用的字符串。能让模拟工作的具体编码模式是：每个字符都表示成一个 0 组成的字符串，其长度与字母表相同，只是在某个位置上有一个 1 反映字符在字母表中的位置。[10]

标签系统字母表里有 4 个字符，所以每个字符都编码成 4 个字符组成的字符串，在不同的位置放上 1：

注 10：0 和 1 的结果序列并不是二进制数，只是含有一个 1 标识特定位置的 0 组成的字符串。

标签系统字符	在字母表中的位置	编码表示
a	0	1000
b	1	0100
c	2	0010
d	3	0001

为了实现这个编码模式，我们将引入 CyclicTagEncoder，它可以由一个特定的字母表构造出来，然后对字母表中的字母进行编码：

```
class CyclicTagEncoder < Struct.new(:alphabet)
  def encode_string(string)
    string.chars.map { |character| encode_character(character) }.join
  end

  def encode_character(character)
  character_position = alphabet.index(character)
    (0...alphabet.length).map { |n| n == character_position ? '1' : '0' }.join
  end
end

class TagSystem
  def encoder
    CyclicTagEncoder.new(alphabet)
  end
end
```

现在可以使用标签系统的 CyclicTagEncoder 对由 a、b、c 和 d 组成的任意字符串进行编码了：

```
>> encoder = system.encoder
=> #<struct CyclicTagEncoder alphabet=["a", "b", "c", "d"]>
>> encoder.encode_character('c')
=> "0010"
>> encoder.encode_string('cab')
=> "001010000100"
```

使用这个编码器，我们可以把每个标签系统规则转换成对应的循环标签系统规则。我们只是对 TagRule 的 append_characters 进行编码，然后使用结果字符串构建一个 CyclicTagRule：

```
class TagRule
  def to_cyclic(encoder)
    CyclicTagRule.new(encoder.encode_string(append_characters))
  end
end
```

在一个 TagRule 上试一下：

```
>> rule = system.rulebook.rules.first
=> #<struct TagRule first_character="a", append_characters="ccdd">
>> rule.to_cyclic(encoder)
=> #<CyclicTagRule "0010001000010001">
```

好，append_characters 已经被转换了，但现在我们已经失去了关于哪个 first_character 应该触发规则的信息——不管它由哪个 TagRule 转换而来，每一个 first_character 都会被字符 1 触发。

此时，该信息由循环标签系统里规则的*顺序*传达：第一个规则针对字母表中的第一个字符，第二个规则针对第二个字符，以此类推。任何在标签系统中没有对应规则的字符都会在循环标签系统中得到一个空规则。

我们可以实现一个 TagRulebook#cyclic_rules 方法返回按照正确顺序排列的转换后的规则：

```
class TagRulebook
  def cyclic_rules(encoder)
    encoder.alphabet.map { |character| cyclic_rule_for(character, encoder) }
  end

  def cyclic_rule_for(character, encoder)
    rule = rule_for(character)

    if rule.nil?
      CyclicTagRule.new('')
    else
      rule.to_cyclic(encoder)
    end
  end
end
```

下面是 #cyclic_rules 为我们的标签系统产生的规则：

```
>> system.rulebook.cyclic_rules(encoder)
=> [
     #<CyclicTagRule "0010001000010001">,
     #<CyclicTagRule "00010001">,
     #<CyclicTagRule "">,
     #<CyclicTagRule "">
   ]
```

转换后的 a 和 b 规则首先出现，后边在 c 和 d 的位置上跟着两个空白规则。

这个结果与模拟工作所依托的字符编码模式相吻合。例如，如果模拟的标签系统的输入字符串是单独的一个字符 b，在循环标签系统的输入字符串中将出现 0100。以下是系统在运行这个输入时的情况：

当前字符串	当前规则	规则可以应用吗
0100	添加字符 0010001000010001 (a 规则)	否
100	添加字符 00010001 (b 规则)	是
00**00010001**	什么都不添加 (c 规则)	否
000010001	什么都不添加 (d 规则)	否
⋮	⋮	⋮

在计算的第一步，当前规则是转换后的 a 规则，并且因为当前字符串以 0 开始，所以当前规则不会应用。但在第二步，随着前导的 0 从当前字符串中被删除，b 规则成为当前规则，同时暴露出一个前导的 1，它将触发规则应用。下两个字符都是 0，因此 c 和 d 规则都不会用到。

可见，通过小心安排输入字符串中字符 1 的出现时间，以便与循环标签系统规则出现的周期一致，我们可以在合适的时间触发合适的规则，完美地模拟常见标签系统规则的字符匹配行为。

最后，我们需要模拟原始标签系统的删除数。这可以通过向循环标签系统的规则手册中插入额外的空规则来完成，以便在一个字符被成功处理后删除合适数量的字符。如果原始的标签系统在其字母表中有 n 个字符，那么原始系统字符串的每一个字符都表示为循环标签系统字符串中的 n 个字符，因此对于每个增加的想要删除的模拟字符，需要 n 个空规则：

```
class TagRulebook
  def cyclic_padding_rules(encoder)
    Array.new(encoder.alphabet.length, CyclicTagRule.new('')) * (deletion_number - 1)
  end
end
```

标签系统的字母表里有 4 个字符，删除数是 2，因此除了已经被转换后规则删掉的字符之外，我们还需要 4 个空规则以删掉一个模拟的字符：

```
>> system.rulebook.cyclic_padding_rules(encoder)
=> [
     #<CyclicTagRule "">,
     #<CyclicTagRule "">,
     #<CyclicTagRule "">,
     #<CyclicTagRule "">
   ]
```

现在我们可以把所有东西都放到一起来为 TagRulebook 实现一个完整的 #to_cyclic 方法，然后在 TagSystem#to_cyclic 方法中使用它，把规则手册和当前字符串都转换成一个完整的循环标签系统：

```
class TagRulebook
  def to_cyclic(encoder)
    CyclicTagRulebook.new(cyclic_rules(encoder) + cyclic_padding_rules(encoder))
  end
end

class TagSystem
  def to_cyclic
    TagSystem.new(encoder.encode_string(current_string), rulebook.to_cyclic(encoder))
  end
end
```

下面是我们转换数字递增标签系统并运行时所发生的：

```
>> cyclic_system = system.to_cyclic
=> #<struct TagSystem ...>
>> cyclic_system.run
100010000100010001000100 (aabbbb) ❶
000100001000100010001000010001000010001
00100001000100010001000010001000010001
0100001000100010001000010001000010001
100001000100010001000010001000010001 (abbbbccdd) ❷
00001000100010001000010001000010001
0001000100010001000010001000010001
001000100010001000010001000010001
01000100010001000010001000010001 (bbbbccdd) ❸
1000100010001000010001000010001 ❹
00010001000100001000100001000100010001
0010001000100001000100001000100010001
010001000100001000100001000100010001 (bbbccdddd)
10001000100001000100001000100010001
0001000100001000100001000100010001
001000100001000100001000100010001
01000100001000100001000100010001 (bbccdddd)
1000100001000100001000100010001 ❺
00010000100010000100010001000100010001
0010000100010000100010001000100010001
010000100010000100010001000100010001 (bccdddddd)
10000100010000100010001000100010001
0000100010000100010001000100010001
000100010000100010001000100010001
00100010000100010001000100010001 (ccdddddd) ❻
0100010000100010001000100010001
100010000100010001000100010001
00010000100010001000100010001 ❼
⋮
001
01
1
❽
=> nil
```

❶ 标签系统的编码后版本的 a 规则在这里。

❷ 模拟字符串的第一个完整字符已经被处理了，因此下面的 4 步使用空规则删除接下来所模拟的字符。

❸ 经过循环标签系统的 8 步之后，所模拟的标签系统完成了完整一步。

❹ 编码后的 b 规则在这里触发了……

❺ ……这里又一次。

❻ 循环标签系统计算 24 步了，而我们到达了所模拟标签系统最终字符串的表示：ccdddddd。

❼ 所模拟的标签系统对于 c 或者 d 开头的字符串没有规则，因此循环标签系统的当前字符串持续变得越来越短……

❽ ……直到变成空字符串，然后系统停机。

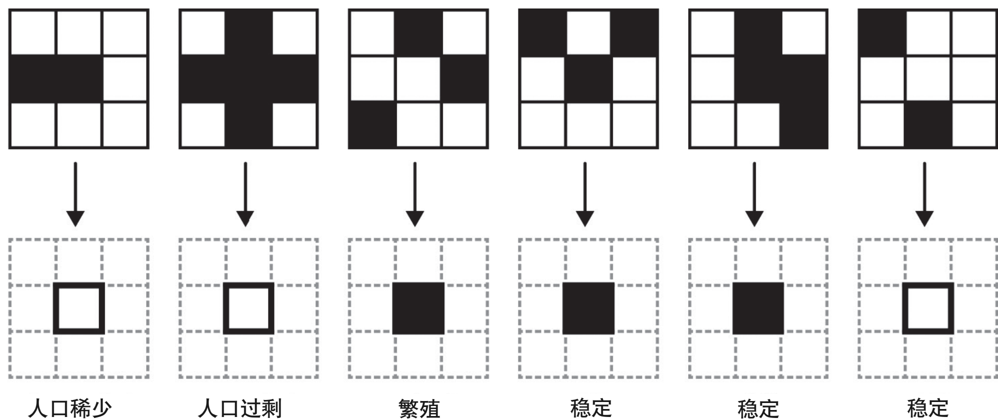
这个技术可以用来模拟任何标签系统，包括本身已经模拟了一台图灵机的标签系统。这意味着循环标签系统也是通用的。

7.7 Conway的生命游戏

1970 年，John Conway 发明了一个叫作*生命游戏*（Game of Life）的通用系统。“游戏”要在一个无限多的二维网格里进行，网格的每个小方格可以是*生*或是*死*。一个小方格有 8 个*邻居*：它上面的三个单元，紧挨着它的左右两个单元，以及它下面的三个单元。

生命游戏像有限状态机那样分一系列步骤进行。在每一步，根据由这个单元自身的当前状态和它邻居的状态所触发的规则，每个单元都可能从生转变为死，或者相反。规则很简单：如果一个活着的单元有少于两个（人口稀少）或者多于三个（人口过剩）活着的邻居，它就会死掉，如果一个死的单元恰好有三个活着的邻居它就能复活（繁殖）。

下面是生命游戏规则如何通过一步的进程来影响一个单元状态的 6 个例子[11]，生的单元用黑色表示，死的单元用白色表示：

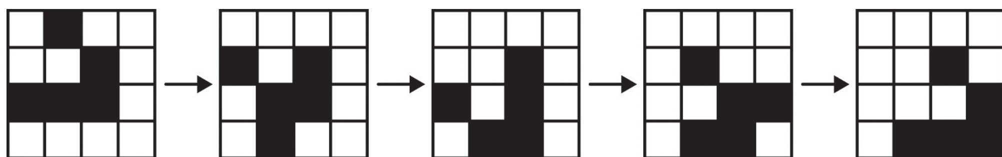




像这样的一个系统，称为细胞自动机，包括一个单元组成的数组和在每一步更新一个单元状态的*规则集合*。

就像本章我们已经看到的其他系统一样，尽管规则简单，但生命游戏展示了出乎意料的复杂性。特定模式的生的单元会出现有趣的行为，其中最著名的就是*滑翔机*（glider），这是一个 5 个生单元的组合，每经过 4 步它们就会沿对角线移动一个方格：

注 11：512 种可能：包括 9 个单元，并且其中每个单元可以是两种状态中的一个，因此有 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2 = 512 种不同的可能。



目前已经发现了很多有意义的模式，包括用不同方式移动的网格形状（spaceship）、产生一连串的其他形状（gun），或者甚至产生它们自身的完整复制品（replicator）。

1982 年，Conway 除了展示如何靠以创造性的方式碰撞“滑翔机”来设计逻辑上的与门（AND）、或门（OR）和非门（NOT）以执行数字计算之外，还展示了如何使用一连串的“滑翔机”来表示二进制数据。这些结构说明理论上可以用生命游戏模拟一个数字计算机，但 Conway 没有设计出来一台可工作的机器。

> 到这里，构造一台任意的大型有限（同时非常慢！）的计算机只是一个工程问题了。我们的工程师已经给出了工具——让他来完成这项工作吧！[……] 我们已经模拟的这种计算机从学术上被称为**通用机器**，因为它可以编程执行任何想要的计算。
>
> ——John Conway，《稳操胜券》（*Winning Ways for Your Mathematical Plays*）

2002 年，Paul Chapman 实现了一个特种通用计算机（http://www.igblan.free-online.co.uk/igblan/ca/）。而 2010 年，Paul Rendell 构造出了一台通用图灵机（http://rendell-attic.org/gol/utm/）。

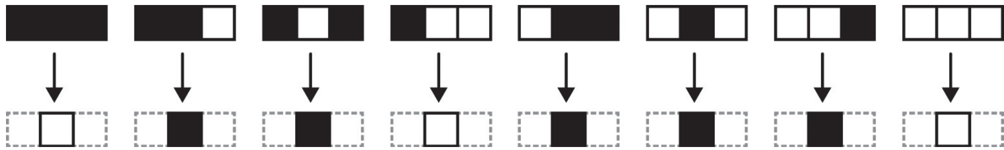
下面是一小部分 Rendell 设计的特写：



7.8 rule 110

rule 110 是另一个细胞自动机，由 Stephen Wolfram 在 1983 年提出。与 Conway 生命游戏里每个单元要么是生的要么是死的类似，rule 110 操作的单元按一维排列而不是二维网格形式。这意味着每个单元只有两个邻居而不是围绕着每个生命游戏单元的 8 个邻居。

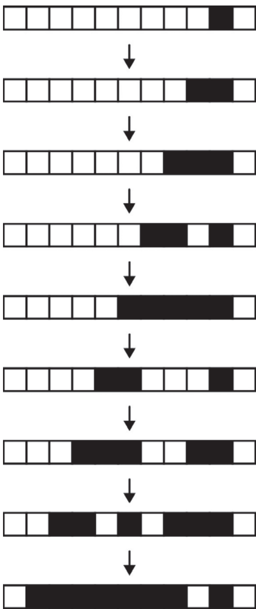
在 rule 110 自动机的每一步，一个单元的下一个状态是由它自身的状态和它两个邻居的状态决定的。与生命游戏里规则都是通用的而且可以应用到生和死单元不同，rule 110 自动机对每一种可能都有一个单独的规则：



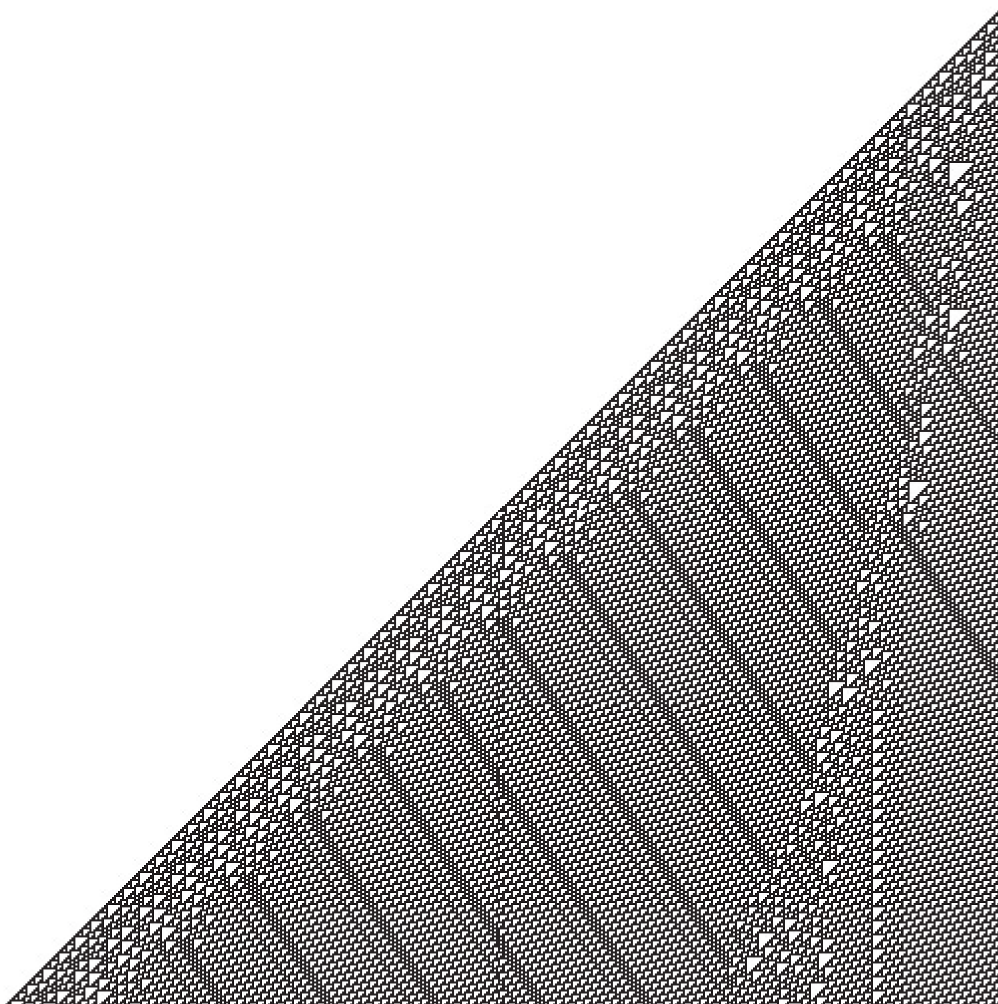


如果我们读取应用这 8 个规则之后的值，把一个死单元当成 0，把一个生单元当成 1，就可以得到二进制数 01101110。再转换可以产生十进制数 110，这就是这个细胞自动机名字的由来。

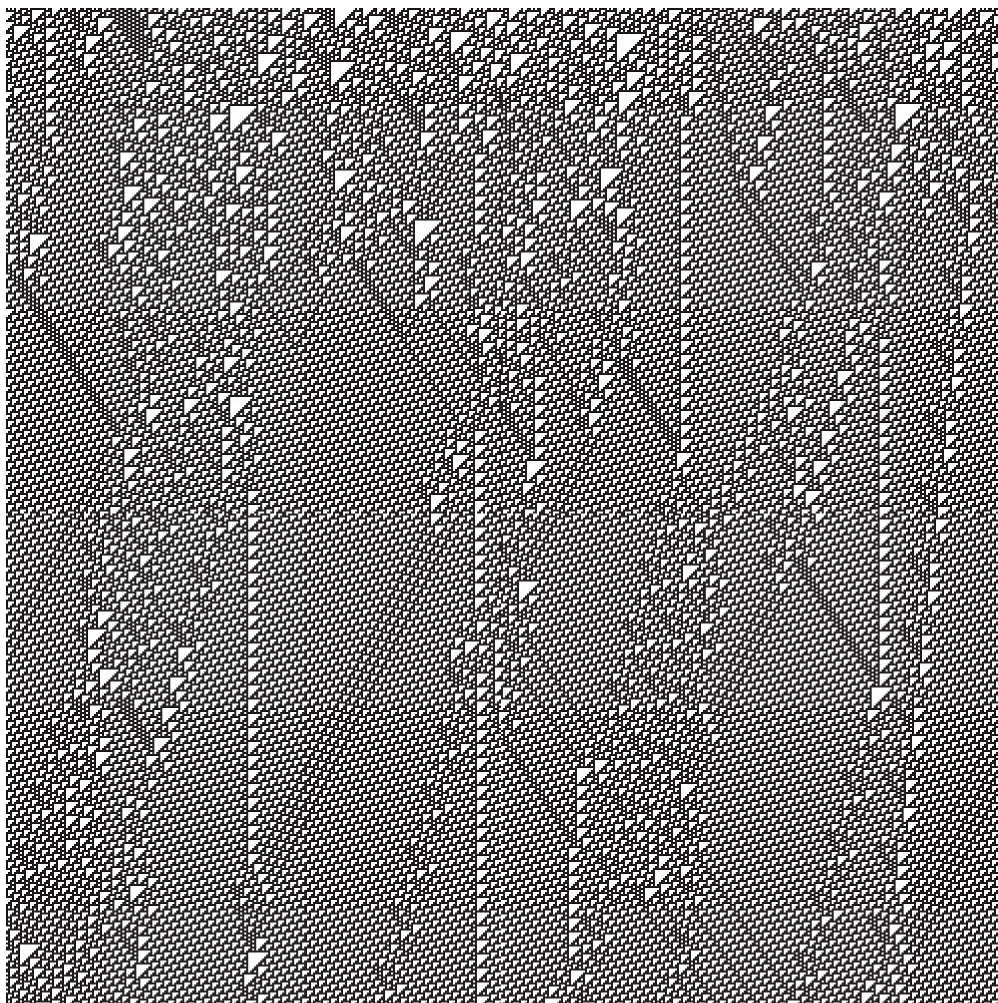
rule 110 比生命游戏简单得多，但它同样有复杂行为的能力。下面是一台 rule 110 自动机从一个简单生单元开始的前几步：



这个行为已经明显不简单了（例如，它不只是在生成一行固定的生单元），而如果运行同样的自动机 500 步，我们就可以看到有趣的模式：



此外，从一个包含生死单元的随机模式开始运行 rule 110，能够揭示所有形状的活动以及它们彼此之间的交互：

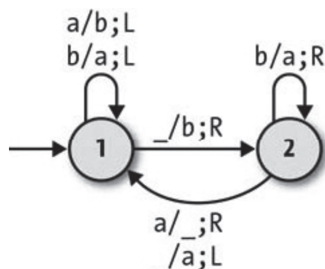


从这 8 条简单规则中浮现出来的复杂性被证明是非常强大的：2004 年，Matthew Cook 发表了一个对 rule 110 事实上通用的证明。这个证明包含大量的细节（参考 http://www.complex-systems.com/pdf/15-1-1.pdf 的第 3 节和第 4 节）。但粗略地讲，它引入了几个不同的 rule 110 模式扮演“滑翔机”的角色，然后通过用一种特定的方式排列那些“滑翔机”来展示如何模拟任意循环标签系统。

这意味着 rule 110 可以运行一个循环标签系统的模拟，而循环标签系统又可以运行一个普通标签系统的模拟，普通标签系统可以运行一个通用图灵机的模拟。这不是完成通用计算的高效方式，但对这样一台简单的细胞自动机来讲仍然是一项令人印象深刻的技术成果。

7.9 Wolfram的2,3图灵机

我们要介绍的最后一个简单通用系统甚至比 rule 110 还简单：Wolfram 的 2,3 图灵机。它的名字源于其两个状态和三个字符（a、b 和空格），这意味着它只有 6 个规则：





这台图灵机与众不同，因为它没有接受状态，因此它从来不会停机，但这主要是一个技术细节。我们仍然可以通过观察特定的行为来得到不停机机器的结果（例如，纸带上一个特定模式字符的出现），并据以认为当前纸带含有有用的输出。

Wolfram 的 2,3 图灵机看起来没有强大到能支持通用计算。2007 年，Wolfram Research 宣布将给予能证明它是通用的人 25 000 美元的奖励。那年下半年，Alex Smith 通过成功的证明拿到了这个奖。就像对 rule 110 一样，这个证明靠的是展示出这种机器可以模拟任何循环标签系统。这个证明还是非常详细的，在 http://www.wolframscience.com/prizes/tm23/ 可以看到全文。

第 8 章

不可能的程序

> 世界上最幸运的事，是人脑无法把自身的内容全部关联起来。
>
> ——霍华德·菲利普·洛夫克拉夫特

本书中，我们已经探索了不同的计算机和编程语言模型，其中包括几种抽象机器。这些机器中有一些更强大，特别是有两种机器有相当明显的限制：有限自动机无法解决涉及无限制计数的问题，例如判定一个括号组成的字符串是否平衡；下推自动机无法处理任何信息需要在多处重用的问题，例如判定一个字符串是否含有同样数目的字符 `a`、`b` 和 `c`。

但我们已经看到的最先进的机器——图灵机，似乎拥有我们需要的一切：拥有无限制的存储，这个存储能以任何顺序、在任意的循环中、在任意的条件语句以及子例程中访问。第 6 章中的极小编程语言 lambda 演算，被证明也出奇得强大：稍加精心设计，它就允许我们把简单的值和复杂的数据结构都表示成纯代码，还能实现操纵这些表示的运算。而在第 7 章，我们看到了许多简单的系统，就像 lambda 演算一样，它们也与图灵机有着同样的通用能力。

我们还能将系统不断增强的过程推进多少？或许并不是不确定的：我们通过增加特性让图灵机做更强大的尝试，但没有取得任何进展，这表明计算能力可能存在着一种硬性的限制。那么计算机和编程语言的基本能力是什么呢？有什么它们做不到的事情吗？存在不可能的程序吗？

8.1 基本事实

这都是相当深奥的问题，因此在试图理解它们之前，我们先回顾一下计算领域的一些基本事实。其中一些事实很明显，而有一些没那么明显，但它们都是思考计算机的能力和限制的前提条件。

8.1.1 能执行算法的通用系统

通常说来，使用像图灵机、lambda 演算和部分递归函数这样的通用系统我们能干什么呢？如果我们能恰当理解这些系统的能力，那就可以考察一下它们的限制。

计算机的实际目的就是执行*算法*。算法是一个指令列表，指令描述把一个输入值转成一个输出值的过程，但必须满足某些条件。

- *有限*

  指令的数量是有限的。

- *简单*

  指令要足够简单，一个人用一支笔和一张纸就能计算出结果。

- *终止*

  对于任何输入，一个遵守指令执行的人都会在有限步骤内终止。

- *正确*

  对于任何输入，一个遵守指令的人都将得到正确的答案。

例如，一个已知最古老的算法是欧几里得算法，这要追溯到公元前 300 年。它以两个正整数为参数，返回能恰好整除它们的最大整数——也就是它们的*最大公约数*。下面是它的指令。

(1) 给定两个数 x 和 y。
(2) 判断 x 和 y 哪个数更大。
(3) 从大的数中减去小的数。（如果 x 更大，就从 x 中减去 y，并把这个新值赋给 x；反之亦然。）
(4) 重复步骤 (2) 和步骤 (3)，直到 x 和 y 相等为止。
(5) x 和 y 相等的时候，它们的值就是原来两个值的最大公约数。

我们很愿意承认这是一个算法，因为它看起来能满足基本的条件。它只包含有限的几条指令，而且都足够简单，对整个问题没有特别理解的人也可以使用铅笔和纸算出结果。再稍微思考一下，我们可以看出对于任意的输入它都一定能在有限步骤内结束：每重复一次步

骤 3，两个数中的一个就会变小一些，因此它们最终一定会到达同样的值[1]并让算法结束。这个算法是否总是能给出正确的答案不是那么明显，但一些代数学的基础就足以证明所得到的结果必定是原始数字的最大公约数了。

所以欧几里得算法确实是一个算法。但像任何算法一样，它只是表示为人类可读语言和符号的思想的集合。如果想要用它做一些有用的事情（或许我们想要探索它的数学性质，或者设计一台自动执行它的机器），我们就需要把算法转换成一个更严格的、歧义更少的形式，这才适合数学分析和机械执行。

我们已经有了一个计算模型用来做这件事情：可以尝试把欧几里得算法写成一台图灵机的规则手册，或者一个 lambda 演算的表达式，或者一个部分递归函数定义，但所有这些都涉及内部的处理以及其他一些枯燥的细节。我们暂时先把它转换成没有限制的 Ruby：[2]

```
def euclid(x, y)
  until x == y
    if x > y
      x = x - y
    else
      y = y - x
    end
  end

  x
end
```

本质上这个 #euclid 方法与欧几里得算法的自然语言描述版本有着同样的指令，但这次它们是用含义严格的定义方式（根据 Ruby 的操作语义）写的，因此可以由一台机器解释：

```
>> euclid(18, 12)
=> 6
>> euclid(867, 5309)
=> 1
```

在这个特定的情况下，很容易把一个非形式化的、人类可读的算法描述转换成对一台机器来说没有歧义的指令。拥有机器可读形式的欧几里得算法非常方便；现在我们无需手工劳动就可以快速可靠地反复执行这个算法了。



很明显我们还可以用与 6.1.7 节类似的技术把这个算法用 lambda 演算来实现，或者从 7.2 节的操作来构建一个部分递归函数，或者像 5.1.2 节那样通过简单算术运算实现一个图灵机的规则集合。

注 1：x 和 y 最小值可以是 1。

注 2：Ruby 已经内建了欧几里得算法 #Integer#gcd，但这不是重点。

这提出了一个重要的问题：任何算法都能转换成适合一台机器执行的指令吗？表面上看，这个问题似乎不值一提——如何把欧几里得算法转换成一个程序相当明显。而作为程序员，我们有天然的倾向会把两者看成可互换的——但在一个计算系统中，一个算法抽象的、直觉的思想与具体的、逻辑上的实现是存在实质差别的。是否存在一个算法，它大、复杂而且不同寻常以致于其本质无法被一个没有思想的机械过程捕捉呢？

最终可能没有严谨的答案，因为这个问题是哲学层面的而非科学层面的。一个算法的指令一定要“简单”而且“不精巧”，以便它“能由一个人计算”，但这些对人类的直觉和能力来说都是不严密的，这并不是能用来证实或者推翻一个假设的数学化断言。

不管怎样，我们都可以通过提出大量算法并观察我们选择的计算系统（图灵机、lambda 演算、部分递归函数，或者 Ruby）是否能够实现它们来收集证据。数学家和计算机科学家差不多从 20 世纪 30 年代开始就已经在这么做了，但到目前为止还没有人成功设计出这些系统不能执行的算法。因此我们可以对经验上的直觉相当自信：一台机器肯定能执行任何算法。

另一个比较强的证据是这些系统中大多数都是为了尝试捕捉和分析一个算法的非形式化思想而独立发展的，只是后来才被发现彼此之间恰好等价。每一次对算法思想的建模尝试都产生了一个系统，这个系统的能力与一台图灵机的能力等价，而这是对一台图灵机足够表示一个算法的很好暗示。

任何算法都能被一台机器（特别是一台确定型的图灵机）执行的思想叫作*邱奇－图灵论题*（Church–Turing thesis）。尽管这仅仅是一个猜想而不是一个被证明的事实，但有足够的证据让它成为广泛接受的真理。



“图灵机能执行任何算法”是个哲学层面的断言，说的是算法的直观感觉和用来实现算法的形式系统之间的关系。它实际的含义是一个解释的问题：我们可以把它看成关于什么能计算以及什么不能计算的命题，或者作为单词“算法”的更严格的一个定义。

不管怎样，它都叫“邱奇－图灵论题”，而不是“邱奇－图灵定理”。因为它是一个非形式化的断言而不是一个可证明的数学断言——它没法用纯数学化的语言表达，因此没有办法构建数学证明。因为它与我们对计算本质的直觉判断和算法能做事情的证据相符，所以被广泛认为是真的，但我们仍旧称它为“论题”，以便提醒自己它的状态与毕达哥拉斯定理这样的可证明思想不同。

邱奇－图灵论题表明，图灵机尽管简单，但拥有执行任何计算所需要的所有能力，而这些计算原则上可以由一个人按照简单的指令执行。许多人比这更进一步，他们认为，既然所有对算法编码的尝试都归结到了与图灵机能力等价的通用系统上，那也就不可能做得更好了：任

何现实世界中的计算机或者编程语言只能做到与图灵机做的一样多的事，不能再多了。是否最终有可能构建一台比图灵机更强大的机器——能使用外来的物理法则执行超越我们对“算法”想象的任务——现在还不能确切知道，但可以肯定的是我们现在不知道如何做。

8.1.2 能够替代图灵机的程序

就像我们在第 5 章中看到的那样，图灵机的简单性使得为一个特定任务设计一个规则手册非常困难。为了避免对可计算性的研究被图灵机编程烦琐的细节干扰，我们将使用 Ruby 程序作为替身，就像处理欧几里得算法那样。

这个方法可行要归因于通用性：原则上，我们可以把任何的 Ruby 程序转换成一个等价的图灵机，反之亦然。因此一个 Ruby 程序与一台图灵机相比不多不少正好能力相当，从而我们发现的关于 Ruby 能力的任何限制都应该可以同样适用于图灵机。

一个明显的异议是 Ruby 有大量的实用函数，而图灵机没有。Ruby 程序可以访问文件系统、发送和接收网络上的消息、接受用户输入、在点阵式显示器上绘图，等等，然而即使最精致的图灵机规则集合也只能在一条纸带上读写。但那不是根本的问题，因为所有这些额外的函数都能用一台图灵机模拟：如果必要，我们可以把纸带的某些部分设计成用来表示“文件系统”或者“网络”或者“显示器”或者任何东西，并把对这些区域的读写处理得就像与外边的真实世界交流一样。这些增强没有一个能改变图灵机的潜在计算能力；它们只是提供了对纸带上活动的高层次的解释。

在实践中，我们可以完全把自己限制在简单的 Ruby 程序避免使用任何有争议的语言特性，以此来规避这个异议。本章的其余部分，我们写程序时将坚持从标准输入中读取，进行一些计算，然后等结束的时候把字符串写到标准输出；输入字符串与一台图灵机纸带的初始内容类似，而输出字符串类似最终的纸带内容。

8.1.3 代码即数据

程序有两种身份。除了把程序当作控制一个特定系统的指令之外，我们还把程序看成是纯数据：一个表达式树，一个原始字符串，或者甚至一个大的数。这种双重性通常会被程序员认为理所当然，但程序能够被表示成数据以便它们能用做提供给其他程序的输入，对通用计算机来说是至关重要的。正是代码和数据的统一才使得软件成为可能。

我们在通用图灵机的讨论中已经看到了作为数据的程序，它期望另一台图灵机的规则手册能作为字符序列写到它的纸带上。像 Lisp[3] 和 XSLT 这样奇特的同体异构编程语言（即程序与数据由同样的结构存储），程序被显式地写成语言本身可以操纵的数据结构：每一个 Lisp 程序是一个称为 *s 表达式*的嵌套列表，而每一个 XSLT 样式表是一个 XML 文档。

注 3：Lisp 实际上是一个编程语言的家族，包括 Common Lisp、Scheme 以及 Clojure，它们有着非常类似的语法。

在 Ruby 当中，通常只有解释器（至少在 MRI 中不是用 Ruby 写的）才会关心程序的结构化表示，但把代码当作数据的原则仍然适用。考虑下面这个简单的 Ruby 程序：

```
puts 'hello world'
```

对于一个熟悉 Ruby 语法和语义的观察者来说，这是一个带上字符串 `'hello world'` 把一个 `puts` 消息发给 `main` 对象的程序，它的执行结果就是 `Kernel#puts` 方法把 `hello world` 进行标准输出。但在更低的层次上，它只是一个字符序列，并且因为字符是表示成字节的，所以最终这个序列可以看成是一个很大的数：

```
>> program = "puts 'hello world'"
=> "puts 'hello world'"
>> bytes_in_binary = program.bytes.map { |byte| byte.to_s(2).rjust(8, '0') }
=> ["01110000", "01110101", "01110100", "01110011", "00100000", "00100111",
    "01101000", "01100101", "01101100", "01101100", "01101111", "00100000",
    "01110111", "01101111", "01110010", "01101100", "01100100", "00100111"]
>> number = bytes_in_binary.join.to_i(2)
=> 9796543849500706521102980495717740021834791
```

从某种意义上说，`puts 'hello world'` 是 Ruby 程序数 9796543849500706521102980495717740021834791。[4] 反过来说，如果某个人告诉我们一个 Ruby 程序的数字，我们很容易把它转换回程序并执行它：

```
>> number = 9796543849500706521102980495717740021834791
=> 9796543849500706521102980495717740021834791
>> bytes_in_binary = number.to_s(2).scan(/.+?(?=.{8}*\z)/)
=> ["1110000", "01110101", "01110100", "01110011", "00100000", "00100111",
    "01101000", "01100101", "01101100", "01101100", "01101111", "00100000",
    "01110111", "01101111", "01110010", "01101100", "01100100", "00100111"]
>> program = bytes_in_binary.map { |string| string.to_i(2).chr }.join
=> "puts 'hello world'"
>> eval program
hello world
=> nil
```

当然，把程序编码成大数是为了把它存储到硬盘上，把它联接互联网，以及把它提供给一个 Ruby 解释器（解释器本身在硬盘上也是一个大数字！），以便让一个特定的计算发生。



既然每一个 Ruby 程序都有一个独一无二的数，那么我们可以自动生成所有可能的程序：从数字 1 开始生成程序，然后生成程序 2，以此类推。[5] 如果用足够长的时间做下去的话，将会最终产生下一个热门的异步 Web 开发框架，然后我们就可以退休颐养天年了。

注 4：只把数字赋值给语法有效的 Ruby 程序会更有用，但那么做会更复杂。

注 5：那些数字中的大多数都不表示语法有效的 Ruby 程序，但我们可以把每个潜在的程序提供给 Ruby 解析器，如果有任何的语法错误的话，就丢弃掉它。

8.1.4 可以永远循环的通用系统

我们已经看到通用目的的计算机是通用的：可以设计一台能模拟其他任何图灵机的图灵机，或者写一个能对其他任何程序求值的程序。通用性是个强大的思想，这样不同的任务只用一台可改写的机器而不是很多专门机器就可以完成。但它也有不方便的地方：任何强大到足以通用的系统，都不可避免地允许我们构建永不停机一直循环的计算。

超长时间运行的计算

> "我想要说的是，"计算机咆哮着，"我的电路现在已经无法撤销地开始计算生命、宇宙和一切终极问题的答案。"它缓了一下，对现在能引起所有人的注意感到很满意，于是降低了音量："但程序运行要稍微花费我一点儿时间。"
>
> 福克不耐烦地瞥了一眼他的手表。
>
> "要多久？"他问。
>
> "750 万年。"深思回答说。
>
> ——道格拉斯 · 亚当斯，《银河系漫游指南》
> （*The Hitchhiker's Guide to the Galaxy*）

如果我们试图执行一个算法——目的是把输入转成输出的指令列表——那么永远循环就是一件坏事了。我们想要一台机器（或者程序）在有限时间内运行然后停机并给出某些输出，而不只是安静地在那儿变热。所有其他都相等的情况下，最好能有计算机和语言，它们的每个任务都保证在有限步骤内结束，这样我们就不必关心最终是否会有答案了。

但是在一些实际的应用中，永远循环是设计好的。例如，一个像 Apache 或者 Ngnix 这样的 Web 服务器如果只能接受一个 HTTP 请求，发送响应然后就退出的话，是没什么用的；我们想要它无限期运行下去，在强制停止前继续为每个到来的请求服务。但从概念上讲，我们可以把一个单线程的 Web 服务器分成两部分：一是处理单个请求的代码，它**应该**总是能停机，以便能发送响应，二是它的外边应该有一个无限循环，能随着每个新请求的到来不断调用请求处理器。在这种情况下，即使封装器需要永远运行，在复杂的请求处理代码里无限循环仍然是一件坏事。

真实世界提供了很多程序的实例，它们在一个无限循环中反复执行停机计算：Web 服务器、GUI 应用、操作系统，等等。尽管我们通常想要算法的输入输出程序总能停机，但这些长时间运行的系统的类似目标是高效，也就是说总是"保持运行"并且永远都不要陷入无响应的状态。

那么为什么每个通用系统都把非终结作为属性呢？有没有什么天才的方法能限制图灵机以便它们总是能停机，而不必在它们的用处上做出妥协呢？怎么知道我们某一天不会设计出一种编程语言，它与 Ruby 一样强大但不包含无限循环呢？对于为什么它们无法做到有各种具体的例子，但还有一个更通用的论据，让我们演练一下。

Ruby 是一种通用编程语言，因此写一个能对 Ruby 代码求值的 Ruby 代码一定是可能的。原则上讲，我们可以定义一个叫 #evaluate 的方法，它的参数是一个 Ruby 程序的代码和一个标准输入提供给程序的字符串，然后对那个程序求值得到结果（也就是说，字符串会发给标准输出）。

在本章中包含进 #evaluate 的实现过于复杂了，但下面是对它最可能工作方式的概括：

```
def evaluate(program, input)
  # 解析程序
  # 在捕获输出的同时基于输入对程序求值
  # 返回输出
end
```

方法 #evaluate 本质上是一个 Ruby 写的 Ruby 解释器。尽管我们还没有对其实现，但写出它来是可能的：首先把程序转成一个符号序列，然后分析它们构建一个解析树（参见 4.3 节），再根据 Ruby 的操作语义（参见 2.3 节）对这个分析树求值。这是一个大而复杂的工作，但它肯定能完成；不然的话，Ruby 就不能满足通用性了。

为了简单，假设我们对 #evaluate 的假想实现是无 bug 的，在它对程序求值的时候不会崩溃。当然它可能会返回某个结果，这个结果表明这个程序在求值的过程中会引发异常，但那与 #evaluate 本身实际执行中的崩溃是不一样的。



Ruby 恰好有一个内建的 Kernel#eval 方法能对 Ruby 代码的字符串求值，但这里利用这个方法有点自欺欺人，特别是因为（在 MRI 中）它是用 C 语言实现的，而不是 Ruby。它对当前的讨论也没有必要；我们把 Ruby 当作任意通用编程语言的典型实例，但许多通用性语言没有内建的 eval。

但是请注意，既然它摆在那儿，为了让 #evaluate 减少一点想象的成分，我们不去用它就太不好意思了。下面是一次粗略的尝试，请多包涵：

```
require 'stringio'

def evaluate(program, input)
  old_stdin, old_stdout = $stdin, $stdout
  $stdin, $stdout = StringIO.new(input), (output = StringIO.new)

  begin
    eval program
  rescue Exception => e
    output.puts(e)
```

```
    ensure
      $stdin, $stdout = old_stdin, old_stdout
    end
    output.string
  end
```

这个实现有许多现实和哲学上的问题，它们都能通过写纯 Ruby 的 #evaluate 来避免。另一方面，从演示角度看，这个实现足够简短而且工作得足够好：

```
>> evaluate('print $stdin.read.reverse', 'hello world')
=> "dlrow olleh"
```

方法 #evaluate 的存在允许我们定义另一个方法：#evaluate_on_itself，它返回用它自己的源代码作为输入对程序求值的结果：

```
def evaluate_on_itself(program)
  evaluate(program, program)
end
```

这可能有点荒唐，但是完全合法；程序只是一个字符串，因此我们完全可以把它既当成一个 Ruby 程序又当成对这个程序的输入。代码即数据，对吧？

```
>> evaluate_on_itself('print $stdin.read.reverse')
=> "esrever.daer.nidts$ tnirp"
```

既然我们知道可以用 Ruby 实现 #evaluate 和 #evaluate_on_itself，因而就能写出完整的 Ruby 程序 does_it_say_no.rb：

```
def evaluate(program, input)
  # 解析程序
  # 在捕获输出的同时基于输入对程序求值
  # 返回输出
end

def evaluate_on_itself(program)
  evaluate(program, program)
end

program = $stdin.read

if evaluate_on_itself(program) == 'no'
  print 'yes'
else
  print 'no'
end
```

这个程序是对现有代码的一个直接应用：它定义了 #evaluate 和 #evaluate_on_itself，然后从标准输入中读取另一个 Ruby 程序，最后把它传给 #evaluate_on_itself。来看看它以自身作为输入的时候程序能干什么。如果输出的结果是字符串 'no'，does_it_say_no.rb 会

输出 'yes'，否则它会输出 'no'。例如：[6]

```
$ echo 'print $stdin.read.reverse' | ruby does_it_say_no.rb
no
```

这是期望的结果；就像我们上面看到的，在用其自身运行 print$stdin.read.reverse 时，会得到输出 esrever.daer.nidts$tnirp，它与 no 不相等。得到输入 no 的程序会怎么样呢？

```
$ echo 'if $stdin.read.include?("no") then print "no" end' | ruby does_it_say_no.rb
yes
```

这次仍然与期望一致。

那么下面是大问题了：在运行 ruby does_it_say_no.rb < does_it_say_no.rb 时，会发生什么呢？[7] 在脑子中要记住 does_it_say_no.rb 是一个真实的程序——用足够的时间和热情可以完整写出来的一个程序——因此，它一定有结果，只是没那么显而易见。让我们试着通过考虑所有的可能然后去掉讲不通的来把它实现出来。

首先，以自身代码作为输入来运行这个特定程序不能产生输出 yes。根据程序自己的逻辑，输出 yes 只能在对自身代码运行 does_it_say_no.rb 输出 no 时才会发生，这与原来的承诺是冲突的。因此这样不行。

好吧，那么可以改为输出 no。但程序的结构意味着，只有同样的计算没有输出 no 它才能输出 no——又冲突了。

有可能输出一些其他字符串，比如 maybe，甚至空字符串吗？那可能还是会冲突：如果 evaluate_on_itself(program,program) 没有返回 no 那程序还是会输出 no。

因此它不能输出 yes 或者 no，不能输出别的什么，并且除非方法 #evaluate 含有 bug，不然它不可能崩溃，但这个已经假定不会了。唯一的可能性是它不产生任何输出，而这只会在程序永不停止的时候才会发生：#evaluate 一定要永远循环，不返回结果。



实际上几乎可以确定 ruby does_it_say_no.rb < does_it_say_no.rb 将会耗尽主机的有限内存，引起 ruby 崩溃，而不会真的永远循环下去。但这是外部施加给程序的资源限制，而不是程序本身的属性；理论上讲，只要有需要我们可以持续给计算机增加更多的内存让计算机无限运行下去。

注 6：我们这里使用的是 Unix shell 语法。在 Windows 平台上，要忽略 echo 参数周围的单引号，或者把文本放到文件里，并把它用 < 输入重定向符提供给 ruby。

注 7：这是一个 shell 命令，以它自身源代码作为输入运行 does_it_say_no.rb。

用这么复杂的方式说明 Ruby 允许我们写不停机程序看起来是没有必要的。毕竟 `while true do end` 能让我们做相同的事，但它简单得多。

但通过思考 does_it_say_no.rb 的行为，我们已经展示了不管系统有什么特性，不停机程序是通用性的一个不可避免的结果。我们的观点除了依赖 Ruby 的通用性之外不依赖 Ruby 的任何特殊能力，因此同样的思想也可以适用于图灵机，或者 lambda 演算，或者任何其他的通用系统。只要在使用一种强大到能对自身求值的语言，我们就知道一定可能使用 `#evaluate` 的等价物构建永不停机的程序，而不需要知道关于语言能力的任何其他东西。

特别地，在编程语言中移除特性（如 `while` 循环）并不能阻止我们在保持语言足以通用的同时还能写出不停机的程序来。如果移除了一个特性让一个程序无法永远循环，一定也不可能实现 `#evaluate` 了。

被仔细地设计以保证它们的程序一定总是能停机的语言叫作*完全编程语言*。与之相对的是更常见的*部分编程语言*，这样语言的程序有时候能停机给出答案，有时候不能。完全编程语言仍然非常强大，能表达许多有用的计算，但它们不能做到的就是解释自身。



这很奇怪，虽然对一种完全编程语言，从定义上来说 `#evaluate` 的等价物一定总是能停机的，但用那种语言是无法实现的——如果它可以实现的话，我们就能使用 does_it_say_no.rb 技术让它永远循环了。

这让我们对一个不可能的程序有了初步了解：无法用完全编程语言写一个对其自身的解释器，即使为了解释它存在一个令人尊敬的保证能停机的算法也不行。事实上，它是如此令人尊敬以至于我们能用另一种更复杂的完全编程语言写出来，但这个新的完全编程语言也不能实现它自己的解释器。

虽然是个有意思的东西，但完全编程语言的设计有人为的限制；我们一直在寻找所有计算机或者编程语言不能完成的东西。我们最好继续努力。

8.1.5　能引用自身的程序

does_it_say_no.rb 使用的自引用的小技巧构建出一个能读自己源代码的程序，但或许假定总是会有点自欺欺人。在我们的例子里，程序收到了自己的源代码作为一个明确的输入，这要感谢环境（如 shell）提供的功能；要没有这个选择的话，它可能也会利用 Ruby 的文件系统 API 和总是包含当前文件名的 `__FILE__` 常量，直接用 `File.read(__FILE__)` 从硬盘读取数据。

但我们应该提出一个通用的论点，只依赖 Ruby 的通用性，而不是依赖操作系统或者 `File` 类的能力。像 Java 和 C 这样运行时没有权限访问自身源代码的编译语言呢？像 JavaScript 这样通过网络连接被加载到内存而且可能根本不会存储到本地文件系统的程序呢？像图灵机和 lambda 演算这样自包含的通用系统，它们根本没有“文件系统”和“标准输入”的

概念，又会怎样呢？

幸运的是，does_it_say_no.rb 参数能经受住这些异议，因为让一个程序从标准输入读取它自己的源代码只不过是一个对所有通用系统都能完成的某个事情的为简化，而且与它们的环境和其他特性无关。这是一个叫作 Kleene *第二递归定理的推论*（Kleene's second recursion theorem），它保证了任何程序都可以转换成能计算自身源代码的等价物。递归理论提供了我们所做简化的合理保证：本可以把 `program = $stdin.read` 用一些代码替换，以便生成 does_it_say_no.rb 的源代码并把它赋给程序而不必进行任何 I/O。

来看看如何在一个简单的 Ruby 程序上做这种转换。例如：

```
x = 1
y = 2
puts x + y
```

我们想要把它转换成类似这样的程序：

```
program = '...'
x = 1
y = 2
puts x + y
```

……这里程序被赋予了一个含有完整程序源代码的字符串。但程序的值应该是多少呢？

一个天真的做法是尝试编造一个能赋值给程序的简单字符串，但这很快就会让我们陷入麻烦，因为这个字符串将是程序源代码的一部分从而会出现在自身的某个地方。这会要求程序以字符串 'program =' 开头，后边是程序的值，这个值还会是字符串 'program ='，后边再跟着程序的值，这样一直类推下去：

```
program = %q{program = %q{program = %q{program = %q{program = %q{program = %q{...}}}}}}
x = 1
y = 2
puts x + y
```



Ruby 的 %q 语法允许我们使用一对定界符来引用不可修改的字符串，在这个场景下是花括号，而不是一对引号。优点是只要定界符能正确匹配，这个字符串就可以包含定界符的非转义实例：

```
>> puts %q{Curly brackets look like { and }.}
Curly brackets look like { and }.
=> nil
>> puts %q{An unbalanced curly bracket like } is a problem.}
SyntaxError: syntax error, unexpected tIDENTIFIER, expecting end-of-input
```

使用 %q 而不是单引号可以帮助我们避免令人头疼的包含自身定界符的字符串里的字符转义：

```
program = 'program = \'program = \\\'program = \\\\\\\'...\\\\\\\'\\\'\''
```

从这个“坑”里爬出来的方法是利用一个事实，那就是一个程序中用到的值没有必要出现在它的源代码里，还可以从其他数据动态计算出来。这意味着我们可以把转换的程序构建成三部分：

A. 把一个字符串赋值给一个变量（如 data）；
B. 使用字符串计算当前程序的源代码并将其赋值给 pragram；
C. 做程序应该做的所有其他工作（原来代码的工作）。

因此，程序的结构将会变成这样：

```
data = '...'
program = ...
x = 1
y = 2
puts x + y
```

这作为一个一般策略听起来貌似有理，但在具体的细节上还有些问题。我们怎么知道 A 部分中要赋值给 data 什么字符串，并且我们怎么用其在 B 部分中对 pragram 进行计算呢？下面是一个解决方案。

- 在 A 部分中，创建一个包含 B 和 C 部分的字符串，并把这个字符串赋值给 data。这个字符串不应该“包含自身”，因为它不是整个程序的源代码，只包含 A 部分之后的部分程序。
- 在 B 部分中，首先计算一个含有 A 部分源代码的字符串。因为 A 部分通常含有一个值可用作 data 的大的字符串，所以我们可以这么做。因此只需要用 'data =' 给 data 的值加上前缀，以此来重建 A 部分的源代码。然后只是把这个结果与 data 连接起来得到整个程序的源代码（因为 data 含有 B 部分和 C 部分的源代码了）并将其赋值给程序。

这个设计仍然有些不够直接（A 部分产生 B 部分的源代码，而 B 部分产生 A 部分的源代码），但它通过保证 B 部分只计算 A 部分的源代码而不必把它包含进来，这刚好避免了无限的倒退。

先把已知的做出来吧。我们已经有了 B 和 C 部分的大部分源代码，因此可以部分地完成数据的值了：

```
data = %q{
program = ...
x = 1
y = 2
puts x + y
}
program = ...
x = 1
y = 2
puts x + y
```



data 需要换行符。通过在一个不可修改的字符串里把这些表示为现行的换行符，而不是表示成可修改的 \n 转义序列，我们就能把 B 和 C 部分的源代码逐字的包括进来，而不必进行任何特殊的编码的转义。[8] 这样直接的复制粘贴让 A 部分的源代码更容易计算。

我们还知道 A 部分的源代码只是字符串 'data = %q{...}'，再加上花括号中间填充好的 data 的值，因此还可以部分地完成 pragram 的值：

```
data = %q{
program = ...
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + ...
x = 1
y = 2
puts x + y
```

现在所有 pragram 中缺失的就是 B 和 C 部分的源代码了，这恰好就是 data 包含的内容，因此我们可以把 data 的值添加到程序来完成任务：

```
data = %q{
program = ...
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
```

最后，回头改进一下 data 的值以反映 B 部分：

```
data = %q{
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
}
program = "data = %q{#{data}}" + data
x = 1
y = 2
puts x + y
```

注 8：因为 B 和 C 部分恰好不包含任何如反斜杠或者不平衡花括号的字符，我们才能绕行成功。如果它们包含的话，我们就得想办法对它们转义然后作为汇编 pragram 值的一部分撤销掉转义。

就是它了！这个程序和原来的作用一样，但现在它有了额外的含有自身代码的本地变量，可它实际上没用那个变量做任何事情。如果转换一个程序，它需要一个程序的本地变量，然后用它做点什么，那会怎么样呢？看下面这个经典的例子：

```
puts program
```

这是一个尝试输出它自己源代码的程序，[9] 但它明显会失败。因为 `program` 是一个未定义的变量。如果我们通过自引用的变换来运行它，可以得到如下结果：

```
data = %q{
program = "data = %q{#{data}}" + data
puts program
}
program = "data = %q{#{data}}" + data
puts program
```

有点意思了。让我们在控制台上看看这个代码能干什么：

```
>> data = %q{
   program = "data = %q{#{data}}" + data
   puts program
   }
=> "\nprogram = \"data = %q{\#{data}}\" + data\nputs program\n"
>> program = "data = %q{#{data}}" + data
=> "data = %q{\nprogram = \"data = %q{\#{data}}\" + data\nputs program\n}\n
program = \"data = %q{\#{data}}\" + data\nputs program\n"
>> puts program
data = %q{
program = "data = %q{#{data}}" + data
puts program
}
program = "data = %q{#{data}}" + data
puts program
=> nil
```

可以确定了，`puts program` 实际上输出了整个程序的源代码。

很明显这个变换不依赖程序本身的任何特别的属性，因此对任何 Ruby 程序它都能工作，而且不必使用 `$stdin.read` 或者 `File.read(__FILE__)` 读取程序自身的源代码。[10] 它也不依赖 Ruby 本身的任何特别属性——只需要像任何其他通用系统一样根据旧值计算新值的能力——这意味着任何图灵机都能引用它自己的编码，任何 lambda 演算表达式都能扩展成含有表示它自身语法的 lambda 演算表达式，以此类推。

注 9：侯世达（Douglas Hofstadter）为输出自己的程序杜撰了名字奎因（quine）。

注 10：是不是忍不住要写一个能对任意 Ruby 程序执行这个转换的 Ruby 程序了？如果使用 `%q{[]` 来引用数据的值，那你如何处理原始代码中的反斜杠和不平衡的大括号呢？

8.2 可判定性

到目前为止我们已经看到图灵机有非常多的能力和灵活性：它们可以执行编码成数据的任意程序，执行我们能想出来的任意算法，运行无限长时间，对它们自身的描述进行计算。尽管它们很简单，可这些小的假想的机器都已经被证明能表示一般的通用系统。

如果它们这么强大而灵活，那是否存在图灵机乃至真实世界的计算机和编程语言不能做的事情呢？

在回答这个问题之前，需要让这个问题更明确一些。我们可以让一台图灵机做什么样的事情呢？怎么识别它已经干完了呢？需要研究每一种可能的问题吗？或者只考虑其中一部分问题是否足够呢？我们只是在寻找解法超越自己当前理解的问题，还是在寻找已经知道永远不能解决的问题呢？

我们可以通过集中在判定性问题上以缩小问题范围。判定性问题的答案为是或者否，就像“2 比 3 小吗？”或者“正则表达式 `(a(|b))*` 与字符串 `'abaab'` 匹配吗？”功能性问题的答案是一个数或者某个非布尔值，如“18 和 12 的最大公约数是多少？”判定性问题比处理功能性问题要容易一些，但它们仍然很有趣，值得我们研究。

如果存在一个算法，对任何可能的输入都能保证在有限时间内解决一个判定性问题，那么这个问题就是*可判定的*（或者叫*可计算的*）。邱奇－图灵论题认为每一个算法都能由图灵机执行，所以对于一个可判定性的问题，我们需要设计一台总是产生正确答案的图灵机，并且如果运行足够长的时间，它总是能停机。把一台图灵机的最终配置解释成“是”或者“否”的答案是很简单的：例如可以检查在当前纸带的位置上是否写有 `Y` 或者 `N`，或者完全忽略纸带内容，而只是检查它的最终状态是接受状态（“是”）还是非接受状态（“否”）。

前几章的所有判定问题都是可判定的。如“有限状态自动机能接受这个字符串吗？”和“这个正则表达式匹配这个字符串吗？”不证自明是可判定的，因为我们已经写了 Ruby 程序以便通过直接模拟有限自动机解决它们。给我们足够的时间和精力，那些程序可以费力地转换成图灵机，而且因为它们的执行包含有限的步骤——DFA 模拟的每一步会消耗输入的一个字符，而输入的是有限数目的字符——它们能保证总是停机给出是或者否的答案来，因此原来的问题都满足可判定的条件。

其他问题有些微妙。“这个下推自动机能接受这个字符串吗？”可能看起来不是可判定的，因为我们已经看到用 Ruby 对一台下推自动机的直接模拟有可能永远循环，也不会给你答案。但是，恰好存在一种方式可以准确地计算出一台特定的下推自动机为了接受和拒绝一

个给定长度的输入字符串要经过多少模拟步骤，[11] 因此问题终究是可判定的：我们只是计算所需要的步数，对那些步骤运行模拟，然后检查输入是否已经被接受了。

那每次都能这么做吗？总是存在一种聪明的方式接近一个问题然后找到一种方法实现一台机器，或者一个程序，让它保证能在有限时间内解决这个问题吗？

好吧，不行，不幸的是不行。有许——*无限多*——多判定性问题而且大量的问题是不可判定的：没有保证能停机的算法能解决它们。这些问题中每一个都是不可判定的，不是因为我们还没有找到合适的算法，而是因为问题本身从本质上就对某些输入不可能解决，而我们可以证明永远也不会找到合适的算法。

8.3 停机问题

大量的非判定性问题是关于机器和程序执行过程中的行为的。这其中最著名的就是*停机问题*，停机问题要解决的是对拥有一条特定纸带的特定图灵机判定它的执行是否能够停机。感谢通用性，我们可以把同样的问题用更实际的名词重讲一遍：给定一个包含 Ruby 程序源代码的字符串，还有一个数据的字符串可以让程序从标准输入中读取，那么运行这个程序最终会得到一个答案作为结果还是只会无限循环下去呢？

8.3.1 构建停机检查器

停机问题应该被看成是不可判定的，尽管原因并不明显。对于一个可回答的问题写出程序是比较容易的。下面是一个不管它的输入字符串是什么，都能确定停机的程序：

```
input = $stdin.read
puts input.upcase
```



我们假设 `$stdin.read` 总是会立即返回一个值——换句话说，每个程序的标准输入是有限的和不会阻塞的——因为我们关注的是程序的内部行为，而不是它与操作系统的交互。

反过来说，对源代码做小小的改动就可以产生一个明显永远不停机的程序：

```
input = $stdin.read

while true
```

注 11：简言之就是：每一台下推自动机都有一个上下文无关文法，反之亦然；任何上下文法都可以用乔姆斯基范式重写；这种范式下的任何上下文无关文法为了生成长度为 n 的字符串一定要经历 $2n-1$ 步。因此我们可以把原始的 PDA 转成一个上下文无关文法，把上下文无关文法重写成*乔姆斯基范式*，然后把这个上下文无关文法转换回 PDA。由此产生的下推自动机与原来的机器能识别同样的语言，但现在我们准确地知道完成它需要多少步了。

```
  # 什么也不做
end

puts input.upcase
```

我们当然可以写出一个停机检查器来区分这两种情况。只是测试程序的源代码是否含有字符串 while true 就够了：

```
def halts?(program, input)
  if program.include?('while true')
    false
  else
    true
  end
end
```

这个 #halts? 方法的实现在下面两个示例程序中会给出正确的答案：

```
>> always = "input = $stdin.read\nputs input.upcase"
=> "input = $stdin.read\nputs input.upcase"
>> halts?(always, 'hello world')
=> true
>> never = "input = $stdin.read\nwhile true\n# do nothing\nend\nputs input.upcase"
=> "input = $stdin.read\nwhile true\n# do nothing\nend\nputs input.upcase"
>> halts?(never, 'hello world')
=> false
```

但 #halts? 对其他程序很可能是错的。例如，存在这样的程序，它们的停机行为依赖于它们的输入值：

```
input = $stdin.read

if input.include?('goodbye')
  while true
    # 什么也不做
  end
else
  puts input.upcase
end
```

因为知道搜索什么，所以我们可以总是扩展停机检查器来处理这样的特殊情况：

```
def halts?(program, input)
  if program.include?('while true')
    if program.include?('input.include?(\'goodbye\')')
      if input.include?('goodbye')
        false
      else
        true
      end
    else
      false
```

```
    end
  else
    true
  end
end
```

现在我们有了一个检查器，它能对三个程序和任意可能的输入字符串给出正确的答案：

```
>> halts?(always, 'hello world')
=> true
>> halts?(never, 'hello world')
=> false
>> sometimes = "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n
# 执行 nothing\nend\nelse\nputs input.upcase\nend"
=> "input = $stdin.read\nif input.include?('goodbye')\nwhile true\n# do nothing\n
end\nelse\nputs input.upcase\nend"
>> halts?(sometimes, 'hello world')
=> true
>> halts?(sometimes, 'goodbye world')
=> false
```

我们可以像这样无限继续下去，增加更多的检查和更多的特殊情况，以支持对实例程序的所有扩展，但我们永远都无法得到判定任意程序是否会停机的全部问题的答案。一个暴力的实现可能会越来越准确，但总是会有盲点；简单的查找特殊语法模式的方法不可能满足所有的程序。

让 #halts? 能在通常情况下对任何可能的程序和输入都工作看起来有些困难。如果一个程序含有任何循环——不管是显式的，如 while 循环，或者隐式的，如递归方法调用——那它都有可能一直运行下去，预测对于给定输入的任何东西都需要对程序含义的熟练分析。作为人类，我们可以立即看出来下面这个程序总是能停机：

```
input = $stdin.read
output = ''

n = input.length
until n.zero?
  output = output + '*'
  n = n - 1
end

puts output
```

但是为什么它总是能停机呢？当然不是因为任何直接的语法原因。解释是 IO#read 总会返回一个 String，而 String#length 总会返回一个非负的 Integer，并且不断对非负的 Integer 调用 -（1）最终总是会产生一个对象，它的 #zero? 方法会返回 true。这个推理链很微妙而且对于小的修改会高度敏感；如果循环中的语句 n=n-1 变成 n=n-2，程序将只会在偶数长度个输入时才会停机。停机检查器需要知道所有这些关于 Ruby 和数的事实，还要知道如何把事实连到一起以便对这种程序的判定能准确。这样的检查器需要

大而复杂。



最基本的困难是不实际执行一个程序很难预测它将会干什么。运行程序 #evaluate 看它是否会停机是很诱人的，但那样做没有好处：如果程序不停机，#evaluate 将会永远运行下去，而不管我们等多久，都不会从 #halts? 获得任何应答。任何可以依赖的停机检测算法都需要在有限的时间内通过检查和分析程序的文本来生成确定的答案，而不是单纯依靠运行程序和等待。

8.3.2 永远不会有结果

好吧，直觉告诉我们 #halts? 很难正确实现，但那并不意味着停机问题是不可判定的。有大量的难题（例如写出 #evaluate）被证明只要付出足够的努力和创造力，都是能解决的。如果停机问题是不可判定的，那就意味着 #halts? 不止是极端困难，而是*不可能*写出来。

如何才能知道 #halts? 的恰当实现不可能存在呢？如果它仅仅是一个工程问题，为什么我们不能投入大量的程序员，并最终获得一个解决方案呢？

1. 好得不真实

我们假设停机问题是可判定的。在这个假想的世界里，写一个 #halts? 的完整实现是可能的，因此对 #halts?(program,input) 的调用在任何 program 和 input 下，总是返回 true 或者 false，并且如果以标准输入的 input 运行，这个答案总是能正确地预测 program 是否能停机。方法 #halts? 的原始结构可能像下面这样：

```
def halts?(program, input)
  # 解析程序
  # 分析程序
  # 如果程序在输入上停机，就返回 true，否则返回 false
end
```

如果可以写 #halts?，那么我们可以构建 does_it_halt.rb，这个程序能读取另一个程序（作为输入），并在读取到空字符串的时候根据那个程序是否停机来输出 yes 或者 no：[12]

```
def halts?(program, input)
  # 解析程序
  # 分析程序
  # 如果程序在输入上停机，就返回 true，否则返回 false
end

def halts_on_empty?(program)
```

注 12：空字符串的选择并不重要；只是任意的一个固定输入。这个设计是在自包含的程序上运行 does_it_halt.rb，程序不从标准输入读取任何东西，因此输入是什么并不重要。

```
  halts?(program, '')
end

program = $stdin.read

if halts_on_empty?(program)
  print 'yes'
else
  print 'no'
end
```

有了 does_it_halt.rb 之后，就可以使用它解决非常难的问题。考虑一下 1742 年克里斯蒂安·哥德巴赫提出的著名论断：

> 任何一个大于 2 的整数都可以写成两个质数之和。

这就是哥德巴赫猜想，因为还没有人能证明它是真还是假，所以它很著名。有证据表明它是真的，因为任选的一个偶数总是可以分成两个质数——12 = 5 + 7、34 = 3 + 31、567 890 = 7 + 567 883，等等——已经检查过它对 4 和 4 000 000 000 000 000 000 之间的所有偶数都成立。但存在无限多个偶数，因此没有计算机能把它们都检查出来，对每个偶数一定可以用这种方式拆分也没有已知的证明。尽管可能性小，但仍有可能存在某个非常大的偶数不是两个质数的和。

证明哥德巴赫猜想是数论的圣杯之一。2000 年，英国费伯出版社悬赏 100 万美元给能证明哥德巴赫猜想的人。但等一下：我们已经有了能发现这个猜想是真的工具了啊！只需要写一个程序，搜索反例即可：

```
require 'prime'

def primes_less_than(n)
  Prime.each(n - 1).entries
end

def sum_of_two_primes?(n)
  primes = primes_less_than(n)
  primes.any? { |a| primes.any? { |b| a + b == n } }
end

n = 4

while sum_of_two_primes?(n)
  n = n + 2
end

print n
```

这在哥德巴赫猜想的真实性和一个程序的停机行为之间建立了联系。如果猜想是真的，这个程序将永远无法找到反例，不管它计数到多少，因此它将会永远循环下去；如果猜想是

假的，n 将最终被赋予一个偶数值，这个偶数值不是两个质数的和，并且程序将会停机。因此我们只需要把它保存成 goldbach.rb 并运行 ruby does_it_halt.rb < goldbach.rb，以查明这是否是一个停机程序，而那将告诉我们哥德巴赫猜想是否是真的。100 万美元是我们的了！[13]

好了，很明显这好得都不真实了。写出能准确预测 goldbach.rb 行为的程序将会要求精通超越我们当前理解的数论知识。数学家已经工作了几百年试图证明或者证伪哥德巴赫猜想；一群贪得无厌的软件工程师构建出一个 Ruby 程序，奇迹般地不止解决这个问题，还能解决可以表达成循环程序的任何未解数学猜想是不可能的。

2. 根本就不可能

到目前为止我们已经看到了很强的证据表明停机问题是不可判定的，但还没有看到确定性的证明。我们的直觉可能是只通过把哥德巴赫猜想转成一个程序就证明或者推翻它是不可能的，但计算有时候是非常违背直觉的，因此我们不应该被多么不可能的东西说服。如果停机问题确实是不可判定的，而不是简单的难以判定，我们应该能够证明它。

下面是为什么 #halts? 永远不能工作。如果它工作，我们就能构建一个新的方法 #halts_on_itself?，这个方法调用 #halts? 以决定一个程序在把它自己的源代码作为输入运行时会做什么：[14]

```
def halts_on_itself?(program)
  halts?(program, program)
end
```

就像 #halts? 一样，#halts_on_itself? 方法总会结束并返回一个布尔值：如果 program 以自己作为输入时能停机就是 true，如果永远循环就是 false。

给定 #halts? 和 #halts_on_itself? 的实现，我们可以写一个叫作 do_the_opposite.rb 的程序：

```
def halts?(program, input)
  # 解析程序
  # 分析程序
  # 如果程序在输入上停机，就返回 true，否则返回 false
end

def halts_on_itself?(program)
  halts?(program, program)
end

program = $stdin.read

if halts_on_itself?(program)
  while true
```

注 13：费伯出版社的奖金在 2002 年过期了，但今天任何能给出证明的人仍然将在明星数学家圈子中名利双收。

注 14：这是对 8.1.4 节中 #evaluate_on_itself 的重现，只是用 #halts? 替换了 #evaluate。

```
    # 什么也不做
  end
end
```

这段代码从标准输入中读取 `program`，查明如果自身为输入时它是否会停机，并立即做相反的动作：如果 `program` 能停机，do_the_opposite.rb 永远会循环；如果 `program` 永远循环，do_the_opposite.rb 会停机。

现在，ruby do_the_opposite.rb < do_the_opposite.rb 会做些什么呢？[15] 就像我们之前用 does_it_say_no.rb 看到的那样，这个问题创造了不可避免的矛盾。

在给定 do_the_opposite.rb 的源码作为参数时，方法 `#halts_on_itself?` 要么返回 `true` 要么返回 `false`。如果它用返回 `true` 表示停机程序，那么 ruby do_the_opposite.rb < do_the_opposite.rb 将会永远循环下去，这意味着 `#halts_on_itself` 是错误的。另一方面，如果 `#halts_on_itself?` 返回 `false`，make do_the_opposite.rb 会立刻停机，又一次与 `#halts_on_itself?` 的预测矛盾。

这里错在选择 `#halts_on_itself?`——它只是一个无辜的小程序，作为 `#halts` 的代码并依赖它的答案。我们真正展示的是在用 do_the_opposite.rb 既作为 `program` 又作为 `input` 的参数时，`#halts?` 不能返回一个满意的答案；不管如何努力工作，它产生的任何结果都是错的。那意味着对于 `#halts?`，任何真正的实现只存在两种可能的命运：

- 给出错误的答案，如即使 do_the_opposite.rb 能停机也预测它永远循环下去（反过来也是这样）；
- 永远循环而且从来也不会返回任何答案，就像 ruby does_it_say_no.rb < does_it_say_no.rb 里 `#evaluate` 做的那样。

因此一个 `#halts?` 完全正确的实现永远不会存在：对于输入，它要么做出错误的预测，要么根本就做不出预测。

回忆一下可判定性的定义：

> 一个判定问题如果存在一个算法能保证对于任何可能的输入都能在有限时间内解决，这个问题就是可判定的。

我们应该证明了写一个 Ruby 程序完全解决停机问题是不可能的，而且既然 Ruby 程序与图灵机等价，所以图灵机也是不可能的。邱奇－图灵论题说的是所有的算法都能由一台图灵机执行，因此如果不存在能解决停机问题的图灵机，也不会存在算法；换句话说，停机问题是不可判定的。

注 15：或者等价地说：如果我们用 do_the_opposite.rb 的源代码作为它的参数调用它，`#halts_on_itself?` 会返回什么呢？

8.4　其他不可判定的问题

能轻松定义的问题，计算机却无法解决，真令人沮丧。但是，这个特定的问题相当抽象，而且我们用来描绘它的 do_the_opposite.rb 程序也不实际而且做作。我们想要 #halts? 实际执行，或者作为一个现实世界应用的一部分写一个 do_the_opposite.rb 的程序看起来不太可能。或许我们可以无视不可判定性，将其作为一个学术“玩具”，然后继续我们的生活。

遗憾的是，没那么简单，因为停机问题不是唯一的不可判定问题。我们日常构建软件的过程中可能想要解决大量问题，而它们的不可判定性对于自动化工具和过程的实际限制非常重要。

来看个小例子。假设我们已经接受了一个任务，要开发一个输出 'hello world' 的 Ruby 程序。听起来相当简单，但按照长期以来的固有模式，我们[16]还要开发一个自动化工具，它能可靠地判定是否存在一个特定的程序在提供一个特定的输入时能输出 hello world。[17]有了这个工具，我们可以分析最终的程序，然后检查它是否做了应该做的事情。

现在，假设我们成功开发了一个方法 #prints_hello_world?，它能正确地对所有程序做出判断。忽略掉实现细节，方法会是这种普遍的形式：

```
def prints_hello_world?(program, input)
  # 解析程序
  # 分析程序
  # 如果程序打印 "hello world"，就返回 true，否则返回 false
end
```

写完最初的程序之后，我们可以使用 #prints_hello_world? 来验证它做了正确的事情；如果做得对，就把它签入到源代码里，发邮件给老板，然后所有人都会很高兴。但情况甚至更好，因为还能使用 #prints_hello_world? 实现另一个有趣的方法：

```
def halts?(program, input)
  hello_world_program = %Q{
    program = #{program.inspect}
    input = $stdin.read
    evaluate(program, input) # evaluate program, ignoring its output
    print 'hello world'
  }

  prints_hello_world?(hello_world_program, input)
end
```

注 16：当然是“负责任的软件工程专业人员”。

注 17：如果程序没有实际从 $stdin 读取任何东西，输入可能是无关的，但为了完整性和一致性我们会把它包含进来。



%Q 语法引用字符串的方式与 %q 一样，之后会执行替换，因此 #{program.inspect} 会被一个包含 program 值的 Ruby 字符串替换掉。

我们新版本的 #halts? 通过构建一个特殊的程序 hello_world_program 来工作，它主要干两件事情：

(1) 用标准输入中的 input 为参数对 program 求值；
(2) 输出 hello world。

hello_world_program 此时执行只有两种可能的结果：要么 evaluate(program, input) 成功结束，在这种情况下 hello world 将会被输出，要么 evaluate(program, input) 将会永远循环，也就根本没有输出。

把这个程序提供给 #prints_hello_world?，以查明那两个结果中哪个将会发生。如果 #prints_hello_world? 返回 true，那意味着 evaluate(program, input) 最终将结束，并允许 hello world 输出，因此 #halts? 返回 true 以标识这个程序对于 input 会停机。相反，如果 #prints_hello_world? 返回 false，那一定是因为 hello_world_program 永远也无法到达它的最后一行，因此 #halts 返回 false，以此来说明 evaluate(program, input) 会永远循环。

我们对 #halts? 的新实现表明停机问题可以规约成检查一个程序是否会输出 hello world 的问题。换句话说，任何计算 #prints_hello_world? 的算法都能改成计算 #halts? 的算法。

我们已经知道一个可工作的 #halts? 不可能存在，因此明显的结论是 #prints_hello_world? 的完整实现也不可能存在。如果不可能实现，邱奇－图灵论题表明不存在这样的算法，因此“这个程序是否会输出 hello world？”是另一个不可判定的问题。

在现实中，没有人关心自动检查一个程序是否会输出特定的字符串，但这个不可判定性证明的结构指向了某种更大更普遍的情况。我们需要构建一个程序，只要其他某个程序停机了，它就展示“print hello world”属性（输出 hello world），这对展示不可判定性足够了。无法重用这种方法的所有程序行为的属性中，有我们确实关心的属性吗？

没有。这是 Rice *定理*：程序行为的任何非平凡性质都是不可判定的，因为停机问题总是能被规约成判定这个属性是否为 true 的问题；如果我们能发明一个算法来判定那个属性，就能使用它来构建另一个算法来判定停机问题，而这是不可能的。



概括地讲，一个“非平凡的属性”是对程序做*什么*而不是程序*怎么*做的一个要求。例如，Rice 定理对于像“这个程序的源代码包含字符串 'reverse' 吗？”这样的问题并不适用，因为这是一个实现细节，能在不改变程序外部可视行为的前提下重构掉。换句话说，像“这个程序是输出它输入的逆向吗？”这样的语义性质是在 Rice 定理范围内的，从而是不可判定的。

Rice 定理告诉我们存在大量关于一个程序执行时会干什么的不可判定的问题。

8.5 令人沮丧的暗示

不可判定性是生命中麻烦的一个事实。停机问题令人失望，因为它表明我们无法拥有一切：我们想要的是能力不受限制的通用编程语言，但还想要写出程序产生一个不会陷入无限循环的结果，或者至少是子例程作为某个更大的长期运行任务的一部分能停机（参见 8.1.4 节“超长时间运行的计算”部分）。

2004 年的一篇经典论文对此做出了简要总结：

> 由于停机问题，语言设计中存在着二分法。根据编程规范，我们必须在这两者间选择。
>
> A. 安全——在这种语言中所有知道的程序都要终止。
> B. 普遍性——在这种语言中，我们可以写：
>
> i. 所有结束的程序；
> ii. 不能结束的病态程序。
>
> 并且，给出一个任意的程序，我们一般无法说出它是（i）还是（ii）。
> 50 年前，在电子计算发展初期，我们选择（B）。
>
> ——David Turner，*Total Functional Programming*（完全函数式编程，http://www.jucs.org/jucs_10_7/total_functional_programming）

是的，我们不愿意写出病态的程序来，但那仅仅是运气不好。没法识别任意的一个程序是否病态，因此我们不可能在不牺牲通用性的前提下完全避免写出病态程序。[18]

Rice 定理的暗示也是令人沮丧的：不止“程序是否会停机”这个问题是不可判定的，“程序是否做了我想让它做的”也是不可判定的。我们生活的宇宙当中，没法构建一台机器能准确预测一个程序是否能输出 `hello world`，是否会计算一个特定的数学函数或者是否能做一个特定的操作系统调用，而这就是它的运行方式。

那是令人沮丧的，因为能够机械地检查程序性质实在是非常有用的；有了一个工具能判定程序是否遵守它的规范或者含有任何的 bug 之后，现代软件的可靠性将会提高。那些性质可能对于个体程序是可以机械地检查出来的，但除非它们通常都能检查出来，不然我们将永远不能信任机器来做这些工作。

例如，假如我们发明了一个新的软件平台，并且决定通过在线商店——一个“应用程序的超市”卖兼容程序来赚钱，如果你喜欢——代表我们平台的第三方开发者。我们想要顾客

注 18：完全编程语言是对这个问题的潜在解决方案，但到目前为止它们还没有开始应用，或许是因为它们比起通常的语言更难理解吧。

能充满自信地购物，因此决定只买满足某些条件的程序：它们一定不能崩溃，它们一定不能调用私有的 API，并且它们一定不能执行从网上下载的任意代码。

成千上万的开发者开始向我们提交代码的时候，我们如何检查每一个应用是否满足要求呢？如果我们使用自动系统检查每一个提交的规范程度，那将会节约大量的时间和金钱，但感谢不可判定性，不可能构建一个准确完成这个任务的系统。我们只能雇用一小队人运行这些程序、反编译并且检测操作系统来测量程序的动态行为，除此之外别无他法。

人工检查速度慢，成本高，容易出错，而且每个程序只能运行一小段时间，提供自己动态行为的有限片段。因此即使没人犯错误，通常一些不可预计的东西也会出现，然后我们就会有大量气愤的顾客。多谢了，不可判定性。

在所有这些不便之下有两个基础问题。第一个是我们没有能力预测程序执行的时候会发生什么；弄清楚一个程序做什么的唯一通用方法就是真正运行它。尽管一些程序足够简单，行为直接是可预测的，但仅仅通过分析它们的源代码，通用语言总是会允许行为不可预测的程序存在。[19]

第二个问题是，在我们确实决定运行程序的时候，没有可靠的方式知道它多久能运行完。唯一通用的解决方案是运行程序然后等它执行，但既然我们知道通用语言的程序有可能不停机永远循环下去，那么总是存在一些程序无论等待多久都运行不完。

8.6 发生上述情况的原因

在这一章里，我们已经看到所有通用系统都足够强大，可以引用自身。程序对数字进行运算，数字可以表示字符串，而一个程序的指令只用字符串写下来的，因此程序完全能够对它们自己的源代码进行运算。

自引用能力使得写出能准确预测程序行为的程序成为不可能的事情。一旦一个特别的行为检查程序写完了，我们总是能构建一个更大的程序打败它：新程序把这个检测器当作一个子例程，检查它自身的源代码，然后立即做与检测器要做的相反的事情。这些自我矛盾的程序比我们实际写出来的一些东西更奇特，但它们只是一个征兆，而不是潜在问题的根因：通常，程序行为过于强大而无法准确预测。



人类语言有类似的能力和问题。“这个句子是一个谎言”（说谎者悖论）是一句话，它不可能是 `true` 也不可能是 `false` 的；就像我们在 8.1.5 节中看到的，任何计算机程序都可以在不需要任何特别语言特性的情况下引用自身。

注 19：Stephen Wolfram 为这种不运行程序就无法预测程序行为的思想起名叫计算不可约。

一言以蔽之，程序行为这么难预测有两个原因。

(1) 任何拥有足够能力引用自身的系统，都无法正确回答每一个关于自身的问题[20]。我们总是可以构建一个像 do_the_opposite.rb 的程序，系统无法预测它的行为。为了避免这个问题，我们需要跳出自引用系统使用一个不同的*更强大*的系统回答关于它的问题。

(2) 但是对于通用编程语言，不存在更强大的系统供我们升级。邱奇－图灵论题表明我们发明的对程序行为进行预测的任何可用算法，都能由一个程序执行，因此我们无法超越通用系统的能力。

8.7　处理不可计算性

写一个程序的所有要点就是让计算机做有用的事情。作为程序员，我们该如何应对无法检测程序是否正确工作这个事实呢？

拒绝是一个吸引人的选择：*忽略整个问题。如果能自动校验程序行为当然好，但我们不能，所以只是期望做到最好，而永远不要检查一个程序在正确地完成它的工作。*

但这属于反应过度，因为情况没有听起来那么坏。Rice 定理并不意味着分析程序不可能，而只是我们不可能写出一个不平凡的*总是*停机并产生正确答案的分析器。就像我们在 8.3.1 节看到的，没有什么可以阻止我们写一个工具来为*某些*程序给出正确答案，只是我们得承认总是会存在*其他*程序要么给出错误答案要么永远循环不返回任何东西。

不考虑不可判定性，下面是一些分析和预测程序行为的实用方法。

- 问一些不可判定的问题，但如果找不到答案就放弃。例如，为了检查一个程序是否会输出特定的字符串，我们可以运行程序然后等待；如果在特定的时间（比如 10 秒）内没有输出那个字符串，我们就结束程序并假设它没有用。我们有可能会扔掉一个 11 秒之后才产生期望输出的程序，但在很多情况下，这种风险是可以接受的，特别是从自身来说我们不需要运行缓慢的程序。
- 把所问的几个小问题答案汇总起来，就能为一个更大的问题提供经验性的证据。在执行自动化验收测试时，我们通常不能为每一个可能的输入检查程序是否做了正确的事情，但我们可以尝试为有限的输入*样本*运行这个程序来看会发生什么。每一个测试运行都对那个特例程序如何运行给出了信息，并且我们可以使用这个信息提高对程序通常可能行为的信心。有可能还有未测试的输入，这会引起完全不同的行为，但只要测试用例为大多数现实输入的表示完成了工作，我们就可以坦然生活。这个方法的另一个例子是单元测试的使用，单元测试是为了验证小段程序行为，而不是

注 20：这大致就是哥德尔第一不完备定理（http://en.wikipedia.org/wiki/G%C3%B6del%27s_incompleteness_theorems 内容。

把程序作为整体来验证。一个良好分离的单元测试专注于简单单元代码的性质，并通过把程序的其他部分表示成测试替代物（存根和模拟对象）来做出假设。使用小段容易理解代码的单个单元测试可能会简单而且快速，把任何一个将会永远运行或者给出误导答案的测试风险最小化。

通过这种方式对程序的所有片段进行单元测试，我们可以建立一个类似数学证明的假设和影响链："如果片段 A 工作，那么片段 B 能工作，而如果片段 B 工作，那么片段 C 能工作。"判定所有这些假设是否正当是人类推理的责任而不是自动化校验的责任。当然，集成和验收测试可以提高我们对整个系统做应做之事的自信。

- 问可判定的问题，在必要的时候要保守一些。上面的建议通过实际运行一个程序的很多部分来看发生了什么，它总是会引入无限循环的风险，但有的问题可以只通过静态检查源代码就能回答。最明显的例子是："这个程序含有任何的语法错误吗？"但万一真正的答案是不可判定的，我们也准备接受近似安全的话，就可以回答更有意思的问题。

  一个常规分析就是浏览程序的源代码看它是否含有计算出来的值从来不用的*死代码*（dead code），或者含有从来不会被求值的*不可达代码*（unreachable code）。我们不可能总能说出是否代码是真正的死代码或者不可达代码，因此只能保守一些，假设它不是，但存在明显是的情况：在某些语言里，我们知道赋值给一个永远不再使用的局部变量肯定是死的，一个紧跟在 `return` 后边的语句必是不可达的。[21] 像 GCC 这样优化的编译器就是使用这个技术识别和去除不必要的代码，让程序更小更快而且不会影响程序的行为。

- 通过把程序转换成更简单的东西来近似它，然后问关于近似的可判定问题。这个重要的思想是下一章的主题。

注 21：Java 语言规范要求编译器拒绝任何含有不可达代码的程序。参见 http://docs.oracle.com/javase/specs/jls/se7/html/jls-14.html#jls-14.21，其中有 Java 编译器如何不运行程序就判定一个程序哪一部分有可能不可达的冗长解释。

第9章

在“玩偶国”中编程

编程就是用语法与机器交流思想。在写程序的时候，我们知道在程序执行的时候我们想要机器做什么，而了解编程语言的语义让我们相信机器将理解程序每个细节的含义。

但复杂的计算机程序远非单个语句和表达式的累加那么简单。一旦把许多小零件组合到一起构成更大的整体，能检查整个程序是否实际做了我们想要它做的事会很有用。例如，我们可能想要知道它总是返回确定的结果，或者运行这个程序能对文件系统或者网络有既定的副作用，或者只是不含有明显的一遇见非期望输入就会导致崩溃的 bug。

实际上，我们可能想要程序拥有各种各样的属性，而如果能只是检查一个特定程序的语法来看它是否有那些属性，将是相当方便的事情。但从 Rice 定理可知，通过看源代码预测一个程序的行为不可能总是给出正确答案。当然，最直接的查明一个程序将会做什么的途径就是执行它，有时候这确实没问题——大量的软件测试就是通过基于已知的输入运行再根据期望的输出检查结果完成的——但有时候运行代码可能也不是一种可接受的方式，原因如下。

首先，任何有用的程序有可能会处理直到运行时才知道的一些信息：来自用户的交互式输入，作为参数传进来的文件，从网络读取的数据，诸如此类的东西。我们当然可以用一些假的输入运行程序以便感知它能做什么，但那只会告诉我们针对这些输入的程序行为，而真正的输入不一样时会发生什么呢？用输入的所有组合运行程序经常是不实际或者不可能的，而用特定集合的输入运行程序虽然可行，却不一定能告诉我们有关其行为的多少信息。

还有一个问题，我们在 8.1.4 节已经探索过了，就是用足够强大[1]的语言写成的程序可以永远运行而从来不会产生结果。这让通过运行程序来可靠地研究任意程序变得不可能，因为有时候不可能预先说出一个程序是否会无限运行（参见 8.3 节），因此任何尝试运行程序的自动监测器都面临永远得不到答案的风险。

最后，即使一个程序不管什么原因，它事先所有的输入数据都可用，而且总是能终止而不会永远循环，运行这个程序的代价也可能非常高或者很不方便。可能会花很长时间才会结束，或者有不可逆转的副作用——发送邮件、汇钱、发射导弹——对于测试的目的，这些都是不应该发生的。

所有这些原因让能够不实际执行程序就能发现它的问题变得很有用。做到这一点的一种方式是使用*抽象解释*，这是一种分析技术。使用这种技术时，我们执行这个程序的简化版本，然后使用执行结果推导出原始程序的性质来。

9.1 抽象解释

抽象解释给了我们一种着手处理难处理问题的方法，这些难处理的问题或许过于庞大，过于复杂，或者有太多的未知东西难以直接处理。抽象解释的主要思想就是使用*抽象*，或者通过让它更小，更简单，或者通过去掉未知的东西，但这样做还能保留足够的细节，以便让它的解决方案与原始问题相关。

为了让这个模糊的想法更具体，让我们看一个抽象解释的简单应用。

9.1.1 路线规划

假设你是一个身处陌生国家的旅行者，想要做到另一个镇的公路旅行计划。你怎么决定要走哪条路线呢？一个直接的解决方案就是跳上你租来的汽车，然后朝看起来最有希望到达目的地的方向行驶。取决于你的幸运程度和外国路标对你的帮助程度，这种对未知道路的暴力探索可能最终让你到达目的地。但这是一个昂贵的策略，而且很可能在完全放弃之前，你会越来越迷路。

使用地图来规划你的旅行是极理性的想法。印在纸上的公路地图是牺牲现实公路网络大量细节之后的一个抽象。它不会告诉你交通是什么样，哪条公路当前关闭了，某个建筑物在哪儿，或者关于第三维的任何东西。至关重要的是，它比真实的东西更小更平。但一张地图确实保留了旅行规划所需要的最重要的信息：所有镇的相对位置，哪条路通向哪个镇，以及哪些路彼此之间如何连接。

尽管丢掉了一些细节，但一个准确的地图仍然是有用的，因为它指定的路线很可能在现实

注 1：“足够强大”这里意思是“通用的”，参见 8.1.4 节。

中是有效的。地图制作人员已经完成了创建现实模型的昂贵工作，这让你能只查看简化的公路网络并规划路线。然后当你驾车驶向目的地时，你可以把计算的结果转换回现实世界中。按照地图这个抽象世界指定的路线，可以避免试错的昂贵代价。

近似的地图让行驶计算更容易，又不会损失结果的准确性。在很多情况下，用地图做决策可能会是错的——无法保证地图告诉你旅行需要的所有信息——但预先规划路线可以让你排除一些错误，让从一个地方到另一个地方容易控制得多。

9.1.2 抽象：乘法的符号

用印刷地图规划路线是抽象解释的现实应用，也非常随意。如果要举一个更正式的例子，我们可以看一下数字的乘法。尽管这仍然是个小例子，但乘法让我们有机会开始写代码研究这些思想。

假设两个数相乘是一个困难或者昂贵的运算，而我们对不实际执行乘法就查明它结果的某些信息很感兴趣。特别地：结果的*符号*是什么？它是一个负数、零，还是一个整数呢？

理论上的难点是在*具体的*世界中进行计算，使用乘法的*标准解释*：真的把数字乘起来，看结果的数，然后决定结果是否为负的，零，或者是正的。例如，在 Ruby 中：

```
>> 6 * -9
=> -54
```

`-54` 是负数，所以我们知道了 `6` 和 `-9` 的乘积是一个负数。任务完成了。

尽管如此，通过在*抽象*世界中进行计算，使用乘法的*抽象解释*，也可能发现同样的信息。就像一个地图使用平面纸上的线来表示现实世界中的道路一样，我们使用抽象的值来表示数字；我们可以在地图上设计一条路线，而不必在真实道路上通过试错来找到路。可以在抽象值上定义一个抽象的乘法运算，而不必使用具体数之上的具体乘法。

为此，我们需要设计抽象的值让计算在结果仍为有用答案的同时，变得更简单。可以利用两个乘数的绝对值 [2] 不影响结果符号的事实：

```
>> (6 * -9) < 0
=> true
>> (1000 * -5) < 0
=> true
>> (1 * -1) < 0
=> true
```

小时候，我们就知道关键要看乘数的符号：两个正数的乘积，或者两个负数的乘积，总是一个正数；一个正数和一个负数的乘积总是负数；而零与任何数的乘积都是零。

注 2：一个数的*绝对值*是把符号去掉时候的值。例如，−10 的绝对值是 10。

因此使用“负数”、“零”和“正数”作为抽象值，可以用 Ruby 定义一个 Sign 类然后创建它的三个实例：

```
class Sign < Struct.new(:name)
  NEGATIVE, ZERO, POSITIVE = [:negative, :zero, :positive].map { |name| new(name) }

  def inspect
    "#<Sign #{name}>"
  end
end
```

这给了我们可以用作抽象值的 Ruby 对象：Sign::NEGATIVE 代表“任何负数”，Sign::ZERO 代表“数字零”，而 Sign::POSITIVE 代表“任意正数”。这三个 Sign 对象组成了这个小的抽象世界，在这个世界里，我们将执行抽象运算。而与此同时，具体的世界里包含着事实上无限个 Ruby 的正数。[3]

我们可以通过实现符号相关的乘法来定义 Sign 值的抽象乘法：

```
class Sign
  def *(other_sign)
    if [self, other_sign].include?(ZERO)
      ZERO
    elsif self == other_sign
      POSITIVE
    else
      NEGATIVE
    end
  end
end
```

Sign 的实例现在可以像数字那样“乘”到一起了，并且 Sign#* 的实现产生的答案与实际数字乘法的一致：

```
>> Sign::POSITIVE * Sign::POSITIVE
=> #<Sign positive>
>> Sign::NEGATIVE * Sign::ZERO
=> #<Sign zero>
>> Sign::POSITIVE * Sign::NEGATIVE
=> #<Sign negative>
```

例如，上面的最后一行问的问题是：我们把任意的正数乘以任意的负数得到的结果是什么？答案是：一个负数。这仍然是一种乘法，但比我们习惯的那种要简单，它只对几乎已经去掉所有识别信息的“数字”起作用。如果把真实的乘法想象成是昂贵的，那这个缩减的乘法版本就是廉价的。

注 3：Ruby 的 Bignum 对象可以表示任意大小的正数，它只受可用内存的限制。

有了数字的抽象世界和对这些数字乘法的抽象解释之后，我们可以用不同的方式处理最初的问题了。我们不是把两个数字直接相乘来找到它们结果的符号，而是把数字转换成它们的抽象表示再把它们相乘。首先，需要一种把具体数转换成抽象数的方法：

```
class Numeric
  def sign
    if self < 0
      Sign::NEGATIVE
    elsif zero?
      Sign::ZERO
    else
      Sign::POSITIVE
    end
  end
end
```

现在，可以转换两个数然后在抽象世界中做乘法了：

```
>> 6.sign
=> #<Sign positive>
>> -9.sign
=> #<Sign negative>
>> 6.sign * -9.sign
=> #<Sign negative>
```

我们又计算出了 `6 * -9` 会得到一个负数，但这次没进行任何实际数字的乘法。步入抽象世界让我们有了执行计算的另一种方式，更重要的是，这个抽象结果能转换回具体的世界，这样就能搞清它的意思，尽管抽象时牺牲细节只得到了一个近似的答案。在这个场景下，抽象结果 `Sign::NEGATIVE` 表明任何具体的数 `-1`、`-2`、`-3` 等都可能是 `6 * -9` 的答案，但答案肯定不是 0 或者任何像 `1` 或 `500` 这样的正数。

注意，因为 Ruby 的值都是对象（带有操作的数据结构），所以可以根据提供的是具体的（`Fixnum`）还是抽象的（`Sign`）对象，我们可以使用同样的 Ruby 表达式为参数执行具体或抽象的计算。用 `#calculate` 方法把三个数用特别的方式乘起来：

```
def calculate(x, y, z)
  (x * y) * (x * z)
end
```

如果使用 `Fixnum` 对象调用 `#calculate`，这个计算将由 `Fixnum#*` 完成，从而得到一个具体的 `Fixnum` 结果。相反，如果我们用 `Sign` 对象调用它，`Sign#*` 操作将会调用并生成一个 `Sign` 结果。

```
>> calculate(3, -5, 0)
=> 0
>> calculate(Sign::POSITIVE, Sign::NEGATIVE, Sign::ZERO)
=> #<Sign zero>
```

这给了我们在真正的 Ruby 程序中执行抽象解释的有限机会，可以把具体的参数替换成它们对应的抽象相对物，然后无需修改就可以运行其他代码了。



这个技术让人联想到自动化单元测试中打桩测试（test doubles）的方法（如存根和模拟对象）。桩是插到代码中的一个特别的占位对象，使用这种方法可以控制和校验代码的行为。在使用更现实的对象作为测试数据特别不方便或者特别昂贵的条件下，它们特别有用。

9.1.3　安全和近似：增加符号

目前为止可以看到，抽象世界中的计算比具体世界中的对应计算在准确性上要差一些，因为抽象会丢掉细节：在地图上规划的路线会表明在哪条路转弯，但不会说在哪条车道行驶，两个 Sign 对象的乘法会表明结果在零的哪一边，但不会告知实际结果值。

很多时候，结果不准确是没问题的，但对一个需要有用的抽象，很重要的是这个不准确是安全的。安全意味着这个抽象总是能给出真相：抽象计算的结果一定要与它对应的具体结果一致。如果不一致，抽象给我们的信息就不可靠，这可能比无用还要差。

Sign 抽象是安全的，因为把数字转换成 Sign，并把它们乘在一起所给出的结果总是与计算数字本身然后把最终结果转成 Sign 一样：

```
>> (6 * -9).sign == (6.sign * -9.sign)
=> true
>> (100 * 0).sign == (100.sign * 0.sign)
=> true
>> calculate(1, -2, -3).sign == calculate(1.sign, -2.sign, -3.sign)
=> true
```

在这方面，Sign 抽象实际上是非常准确的。它准确保留了合适数量的信息并通过抽象计算把它们完美保留下来。在抽象与想要执行的计算不是那么匹配的时候，安全性问题变得更重要了，通过抽象加法实验我们将看到这一点。

两个数的符号如何确定它们加到一起得到的数字的符号，有一些规则，但它们并不是对所有可能的符号组合都有作用。我们知道两个正数的和一定是正数，而一个负数和零的和一定是负数，但如果把一个负数和一个正数加到一起会怎么样呢？在这种情况下，结果的符号取决于两个数绝对值的关系：如果正数的绝对值比负数的绝对值大，我们得到的答案就是正的（−20+30=10），如果负数的绝对值更大，那就会得到负数的答案（−30+20=−10），而如果它们的绝对值恰好相等，会得到零。但当然，每个数的绝对值正好是我们的抽象已经丢弃的信息，因此不可能在抽象世界中做出这种符号的判定。

对我们的抽象这是一个问题，因为它太抽象了，不能在每种情况下都准确地进行计算。如何处理这种情况呢？我们可以添加抽象加法的定义让它返回同样的结果——比如说只要不知道正确答案的时候就返回 Sign::ZERO——但那会不安全，因为那意味着抽象计算给出的答案可能与通过具体计算得到的答案不一致。

解决方案就是扩展抽象以适应这个不确定性。就像 Sign 值意思是“任何正数”和“任何负数”一样，我们可以引入一个新的，它只表示“任何数”。这实际上是最实在的答案，在遇到问题但没有足够细节的时候我们可以给出这个答案来：结果可能是负数、零，或者正数，不保证到底是哪种。让我们管这个新值叫作 Sign::UNKNOWN：

```
class Sign
  UNKNOWN = new(:unknown)
end
```

这给了我们安全实现抽象加法所需要的东西。计算两个数 x 和 y 之和的符号的规则是：

- 如果 x 和 y 符号相同（同为正、同为负，或者都是零），那这个符号就是它们和的符号；
- 如果 x 是零，它们的和与 y 的符号相同，反过来也是这样；
- 否则，它们和的符号未知。

很容易把这些规则转换成 Sign#+：

```
class Sign
  def +(other_sign)
    if self == other_sign || other_sign == ZERO
      self
    elsif self == ZERO
      other_sign
    else
      UNKNOWN
    end
  end
end
```

这样给出的行为正是我们想要的：

```
>> Sign::POSITIVE + Sign::POSITIVE
=> #<Sign positive>
>> Sign::NEGATIVE + Sign::ZERO
=> #<Sign negative>
>> Sign::NEGATIVE + Sign::POSITIVE
=> #<Sign unknown>
```

事实上，在输入中有一个符号未知的时候这个实现恰好做了正确的事情：

```
>> Sign::POSITIVE + Sign::UNKNOWN
=> #<Sign unknown>
>> Sign::UNKNOWN + Sign::ZERO
```

```
=> #<Sign unknown>
>> Sign::POSITIVE + Sign::NEGATIVE + Sign::NEGATIVE
=> #<Sign unknown>
```

但是我们确实需要回去修改 Sign#* 的实现，以便它能正确地处理 Sign::UNKNOWN：

```
class Sign
  def *(other_sign)
    if [self, other_sign].include?(ZERO)
      ZERO
    elsif [self, other_sign].include?(UNKNOWN)
      UNKNOWN
    elsif self == other_sign
      POSITIVE
    else
      NEGATIVE
    end
  end
end
```

这样我们就有了两个可以使用的抽象操作。注意，Sign::UNKNOWN 是不传染的，即使一个未知数乘以零也仍然是零，因此任何中间存在的不确定性都可能在结束时被消化掉：

```
>> (Sign::POSITIVE + Sign::NEGATIVE) * Sign::ZERO + Sign::POSITIVE
=> #<Sign positive>
```

为了处理 Sign::UNKNOWN 引入的不准确性，我们还需要调整对正确性的认识。因为抽象有时候没有足够的信息给出准确答案，一个计算的抽象和具体版本也不总是能给出互相准确匹配的结果了：

```
>> (10 + 3).sign == (10.sign + 3.sign)
=> true
>> (-5 + 0).sign == (-5.sign + 0.sign)
=> true
>> (6 + -9).sign == (6.sign + -9.sign)
=> false
>> (6 + -9).sign
=> #<Sign negative>
>> 6.sign + -9.sign
=> #<Sign unknown>
```

怎么回事呢？抽象还安全吗？是的，因为在失去准确度返回 Sign::UNKNOWN 的时候，抽象计算告诉我们的仍然是某种事实："结果是一个负数、零，或者正数。"它没有执行具体计算所得到的结果有用，但它没错，并且它好在没有往抽象值中添加更多信息从而让抽象计算变复杂。

我们在代码中可以用一种比 #== 更好的方式来比较符号，#== 现在太不利于安全检查了。这里想要知道的是：具体计算的结果在抽象计算所预测的结果*范围内*吗？如果抽象计算声称可能有几个不同的结果，那具体计算是实际产生了这个结果中的一个，还是完全是另外

的结果呢？

在 Sign 上定义一个操作，它可以告诉我们两个抽象值是否用这种方式彼此关联。既然我们在测试一个 Sign 的值是否“落在”另一个里，那么叫它 #<= 方法吧：

```
class Sign
  def <=(other_sign)
    self == other_sign || other_sign == UNKNOWN
  end
end
```

这样我们就可以做测试了：

```
>> Sign::POSITIVE <= Sign::POSITIVE
=> true
>> Sign::POSITIVE <= Sign::UNKNOWN
=> true
>> Sign::POSITIVE <= Sign::NEGATIVE
=> false
```

现在可以检查安全性了，看一下是否每个具体计算的结果都落在了抽象计算预测的范围里：

```
>> (6 * -9).sign <= (6.sign * -9.sign)
=> true
>> (-5 + 0).sign <= (-5.sign + 0.sign)
=> true
>> (6 + -9).sign <= (6.sign + -9.sign)
=> true
```

安全性对包括加法和乘法在内的任何计算都能保持，因为当抽象计算无法给出准确答案的时候，我们已经设计了一个能进行安全近似的抽象。

顺便说一下，能访问这个抽象让我们能对进行数的加和乘的 Ruby 代码做简单的分析。作为一个实例，下面是一个计算平方和的方法：

```
def sum_of_squares(x, y)
  (x * x) + (y * y)
end
```

如果想要自动分析这个方法以了解它的某些行为，我们可以把它处理成黑盒，用所有可能的参数运行它，这可能会造成永久运行；也可以检查它的源代码并尝试使用数学推理来推导出它的属性，这样很复杂。（而在一般情况下，由于 Rice 定理这注定失败。）抽象解释给了我们第三个选项，可以用抽象值调用这个方法，看这个计算的抽象版本会产生什么输出，因为抽象值的组合数只是一个很小的数字，所以为所有的可能输入这么做也是可行的。

每个参数 x 和 y 都可能是负数、零或者正数，因此让我们看看输出都有哪些可能：

```
>> inputs = Sign::NEGATIVE, Sign::ZERO, Sign::POSITIVE
=> [#<Sign negative>, #<Sign zero>, #<Sign positive>]
>> outputs = inputs.product(inputs).map { |x, y| sum_of_squares(x, y) }
=> [
     #<Sign positive>, #<Sign positive>, #<Sign positive>,
     #<Sign positive>, #<Sign zero>, #<Sign positive>,
     #<Sign positive>, #<Sign positive>, #<Sign positive>
   ]
>> outputs.uniq
=> [#<Sign positive>, #<Sign zero>]
```

不必经过任何智能分析，这就能告诉我们 `#sum_of_squares` 只能产生零或者正数，从来不会有负数——对于读过代码的人来说，这是一个相当无聊的特性，但对机器来说，这都无所谓。当然，这种小技巧只对非常简单的代码起作用，但尽管是个小玩具，它还是展示了抽象如何能让一个难题变得更容易处理。

9.2　静态语义

到目前为止，我们已经看到了如何不实际执行计算就能发现它的近似信息。我们本可以通过实际执行计算来获得更多信息，但近似的信息比没有还是要强，而且对于某些程序（如路线规划），可能这就是我们所需要的全部了。

在乘法和加法的例子里，我们通过把输入的具体数换成抽象值，把一个小程序转成了一个更简单更抽象的版本，但如果想要研究更大更复杂的程序，用这种技术只能到这个程度了。提供给它们自己乘法和加法实现的值很容易创建，但更一般的情况下，Ruby 并不允许值控制它们自身的行为（例如在 `if` 语句中使用它们的时候），因为它对特定的语法片段如何工作有硬编码的规则[4]。除此之外，仍然存在的问题是：因为一些程序会永远循环而不会返回结果，所以通常情况下通过运行程序并等待其输出来了解程序并不可行。

乘法和加法的例子还有另一个缺点，那就是它们没什么意思，没有人会关注它们的程序返回正数或者负数。在实践中，有意思的是像“我的程序运行时会崩溃吗？”和“我的程序能变得更有效率吗？”这类问题。

我们可以通过思考它们的*静态语义*来回答关于程序的更有趣的问题。在第 2 章，我们了解了编程语言的*动态语义*，一种定义代码运行时含义的方法。一种语言的静态语义告诉我们程序性质，无需执行就可以研究。静态语义的经典例子就是*类型系统*：它是一个能用来分析程序的规则集合，能检查其中是否含有某种 bug。在 2.3.1 节的“正确性”里，我们考虑的是像 «`x=true; x=x+1`» 这样的 Simple 程序，它在语法上有效但执行时会引起动态语义的问题。一个类型系统可以事先预判这些错误，在一些坏程序被人尝试执行之前就自动拒绝它。

注 4：和 SmallTalk 不同。

抽象解释给了我们思考程序静态语义的方式。程序注定要执行，因此一个程序含义的标准解释就是由它的动态语义给出的：«x=1+2; y=x*3» 这个程序通过进行算术运算并把它们存储在内存的某个地方来操纵数字。但如果有另一个这种语言的更抽象语义，我们可以根据不同的规则"执行"同样的程序，并得到更抽象的结果，这个结果可以提供关于程序在正常解释时所发生事情的一部分信息。

9.2.1 实现

通过为第 2 章的 Simple 语言构建一个类型系统，我们可以把这个思想具体化。表面上，这看起来很像 2.3.2 节中的大步操作语义：将为每个表示 Simple 程序（Number、Add 等）的语法类实现一个方法，而且调用这个方法将会返回一个最终结果。在动态语义中，这个方法叫 #evaluate，而且它的结果要么是完全求过值的 Simple 值，要么是一个把名字和 Simple 值关联起来的环境，这取决于是在对表达式求值还是在对语句求值：

```
>> expression = Add.new(Variable.new(:x), Number.new(1))
=> «x + 1»
>> expression.evaluate({ x: Number.new(2) })
=> «3»
>> statement = Assign.new(:y, Number.new(3))
=> «y = 3»
>> statement.evaluate({ x: Number.new(1) })
=> «:x=>«1», :y=>«3»}
```

对于静态语义，我们将实现不同的方法，它做的工作更少而且会返回更抽象的结果。这里的抽象值不是具体的值和环境，而是*类型*。一个类型代表许多可能的值：一个 Simple 表达式可以求值成一个数或者一个布尔值，因此对于表达式，我们的类型将是"任何数"和"任何布尔值"。这些类型与之前看到的 Sign 值类似，特别是像实际上含义是"任何数"的 Sign::UNKNOWN。就像 Sign 那样，可以通过定义一个叫 Type 的类并创建一些实例来引入类型：

```
class Type < Struct.new(:name)
  NUMBER, BOOLEAN = [:number, :boolean].map { |name| new(name) }

  def inspect
    "#<Type #{name}>"
  end
end
```

新方法将会返回类型，因此我们叫它 #type。它应该回答一个问题：这个 Simple 语法求值的时候，它将返回哪种类型的值呢？这对 Simple 的 Number 和 Boolean 语法类很容易实现，因为数字和布尔值求值之后为自身，因此我们能准确地知道将得到的值的类型：

```
class Number
  def type
    Type::NUMBER
```

```
  end
end

class Boolean
  def type
    Type::BOOLEAN
  end
end
```

对于像 Add、Multiply 和 LessThan 这样的操作，就要复杂一点了。例如，我们知道对 Add 求值会返回一个数，而我们还知道只有 Add 的两个参数都求值为一个数时它才能求值成功，不然 Simple 解释器将会报错：

```
>> Add.new(Number.new(1), Number.new(2)).evaluate({})
=> «3»
>> Add.new(Number.new(1), Boolean.new(true)).evaluate({})
TypeError: true can't be coerced into Fixnum
```

怎么弄清楚一个参数是否将求值成一个数呢？那是它的类型告诉我们的。因此对于 Add，规则类似这样：如果两个参数的类型是 Type::NUMBER，那最终的结果类型是 Type::NUMBER；不然的话，结果没有类型，因为任何试图进行非数字加法的表达式求值都会在返回任何结果之前失败。为了简单，我们将让 #type 方法返回 nil 以表明这个失败；在其他环境下，如果能让最终的实现更简单，我们可能会选择抛出异常或者返回某个特别的错误值（例如 Type::ERROR）。

Add 的代码看起来像这样：

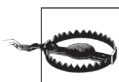
```
class Add
  def type
    if left.type == Type::NUMBER && right.type == Type::NUMBER
      Type::NUMBER
    end
  end
end
```

对 Multiply#type 的实现是一样的，LessThan#type 也非常类似，只是它会返回 Type::BOOLEAN 而不是 Type::NUMBER：

```
class LessThan
  def type
    if left.type == Type::NUMBER && right.type == Type::NUMBER
      Type::BOOLEAN
    end
  end
end
```

在控制台上，我们可以看到这足以区分能成功求值和不能成功求值的表达式，而 Simple 的语法两者都支持：

```
>> Add.new(Number.new(1), Number.new(2)).type
=> #<Type number>
>> Add.new(Number.new(1), Boolean.new(true)).type
=> nil
>> LessThan.new(Number.new(1), Number.new(2)).type
=> #<Type boolean>
>> LessThan.new(Number.new(1), Boolean.new(true)).type
=> nil
```



我们假设抽象语法树至少句法上是有效的。由于树叶子上的实际值被静态语义忽略了，所以 #type 可能会错误预测一个坏形式表达式的求值行为：

```
>> bad_expression = Add.new(Number.new(true) Number.new(1)) ❶
=> «true + 1»
>> bad_expression.type
=> #<Type number> ❷
>> bad_expression.evaluate({})
NoMethodError: undefined method `+' for true:TrueClass ❸
```

❶ 这个抽象语法树的高层结构看起来正确（一个 Add 含有两个 Number），但第一个 Number 对象是畸形的，因为它的值属性是 true 而不是 Fixnum。

❷ 静态语义假设把两个 Number 加在一起总是产生另一个 Number，因此 #type 说求值将会成功……

❸ ……但如果实际对这个表达式求值，在 Ruby 尝试往 true 上加 1 的时候我们会得到一个异常。

Simple 解析器应该永远也不会产生坏形式的表达式，因此这在实际中不太可能是问题。

这是之前加法、乘法和 Sign 小技巧的更通用的版本。即使没有进行任何实际的加法或者数字比较，静态语义给了我们“执行”程序的另一种方式，这种方式仍将返回有用的结果。

我们没有把表达式 «1+2» 解释成关于值的程序，而是扔掉一些细节，把它解释成关于类型的一个程序，而静态语义提供了 «1»、«2» 和 «+» 的另一种解释，这让我们运行这个关于类型的程序来看看结果是什么。这个结果没那么具体，比起我们根据动态语义正常运行程序所得到的更抽象，但尽管如此它仍然是个有用的结果，因为我们有办法把它转换成具体世界中有意义的一些东西：Type::NUMBER 意味着“在这个表达式上调用 #evaluate 将会返回一个 Number”，而 nil 的意思是“调用 #evaluate 可能会引起错误”。

我们现在几乎有了 Simple 表达式的完整静态语义，但还没看变量呢。Variable#type 应该返回什么呢？这取决于变量含有什么值：在像 «x=5; y=x+1» 的程序里，变量 y 拥有类型 Type::NUMBER，但在 «x=5; y=x<1» 里，它的类型是 Type::BOOLEAN。怎么处理这种情况呢？

我们在 2.3.1 节中看到，Variable 的动态语义使用一个环境散列把变量名映射到它们的值上，而静态语义需要某种类似的东西：从变量名到*类型*的映射。我们可以称其为“类型环境”，但还是使用*类型上下文*这个名称以便避免与这两种环境混淆。如果把一个类型上下文传给 Variable#type，它需要做的就是在上下文中查找这个变量：

```
class Variable
  def type(context)
    context[name]
  end
end
```



这个类型上下文来自哪里呢？目前，我们将假设它能通过某种方式得到，不管什么时候需要，都能通过某种外部机制提供。例如，或许每个 Simple 程序都有一个头文件来声明所有用到的变量；这个文件在程序运行的时候没有作用，而只是用来在开发过程中与静态语义进行自动检查。

现在 #type 期望一个上下文参数，我们需要回过头去修改出 #type 的另一个实现以接受一个类型上下文：

```
class Number
  def type(context)
    Type::NUMBER
  end
end

class Boolean
  def type(context)
    Type::BOOLEAN
  end
end

class Add
  def type(context)
    if left.type(context) == Type::NUMBER && right.type(context) == Type::NUMBER
      Type::NUMBER
    end
  end
end

class LessThan
  def type(context)
    if left.type(context) == Type::NUMBER && right.type(context) == Type::NUMBER
      Type::BOOLEAN
    end
  end
end
```

这提供了包含变量的表达式类型，只要给它们提供一个正确类型的上下文即可：

```
>> expression = Add.new(Variable.new(:x), Variable.new(:y))
=>«x + y»
>> expression.type({})
=> nil
>> expression.type({ x: Type::NUMBER, y: Type::NUMBER })
=> #<Type number>
>> expression.type({ x: Type::NUMBER, y: Type::BOOLEAN })
=> nil
```

这给了我们各种表达式语法的 #type 实现，那么语句呢？对一个 Simple 语句求值会返回一个环境，而不是一个值，那么在静态语义中如何表达呢？

处理语句的最简单方式就是把它们看成是一种无效的表达式：假设它们不返回值（这是真的）并且忽略它们对环境的影响。我们可以想出一个含义是“不返回值”的新类型，并把这个类型与任何子部件有正确类型的语句联系起来。给这种新类型起名字叫 Type::VOID：

```
class Type
  VOID = new(:void)
end
```

DoNothing 和 Sequence 的 #type 实现很简单。DoNothing 的求值总是会成功，只要连接的语句没有错误，对 Sequence 的求值就会成功：

```
class DoNothing
  def type(context)
    Type::VOID
  end
end

class Sequence
  def type(context)
    if first.type(context) == Type::VOID && second.type(context) == Type::VOID
      Type::VOID
    end
  end
end
```

If 和 While 则都含有能作为条件的表达式，而且为了让程序能工作正常，这个条件必须求值成一个布尔值：

```
class If
  def type(context)
    if condition.type(context) == Type::BOOLEAN &&
      consequence.type(context) == Type::VOID &&
      alternative.type(context) == Type::VOID
      Type::VOID
    end
  end
end
```

```
class While
  def type(context)
    if condition.type(context) == Type::BOOLEAN && body.type(context) == Type::VOID
      Type::VOID
    end
  end
end
```

这让我们能区分求值过程中会出错和不会出错的语句：

```
>> If.new(
     LessThan.new(Number.new(1), Number.new(2)), DoNothing.new, DoNothing.new
   ).type({})
=> #<Type void>
>> If.new(
     Add.new(Number.new(1), Number.new(2)), DoNothing.new, DoNothing.new
   ).type({})
=> nil
>> While.new(Variable.new(:x), DoNothing.new).type({ x: Type::BOOLEAN })
=> #<Type void>
>> While.new(Variable.new(:x), DoNothing.new).type({ x: Type::NUMBER })
=> nil
```



Type::VOID 和 nil 在这里有不同的含义。#type 返回 Type::VOID 的时候，意思是“这个代码很好只是没设返回值”；nil 意思是“这个代码含有错误。”

唯一还没实现的方法就是 Assign#type。我们知道它应该返回 Type::VOID，但在什么环境下呢？如何决定一个赋值行为是否良好呢？想要根据静态语义检查赋值语句右侧的表达式是否合理，但关心它是什么类型吗？

这些问题让我们要对什么应该是有效的 Simple 程序做出一些设计决策。例如，«x=1; y=2; x=x<y» 可以吗？根据动态语义它当然没问题——在它执行的时候不会发生什么坏事——但我们可能（或者可能不！）对变量在执行中从持有一种类型的值转为持有另一种类型的值感到不舒服。这种灵活性可能对一些程序员有价值，但对其他人则可能是意外错误的来源。

从设计静态语义的人的角度来说，处理一种变量类型可以改变的语言也更困难。到目前为止我们假设类型的上下文来自外部并在整个程序中不做改变。但可以选择一个更复杂的系统，这个系统的上下文在程序的开头是空的，而上下文随着变量的声明和赋值逐渐构建起来。这种方式与随着程序的执行动态语义逐渐构建起值的环境一样。但这样很复杂：如果语句能改变类型上下文，那将需要 #type 方法既返回一个类型又返回一个上下文，这种方式与动态语义的 #reduce 方法返回一个规约的程序和一个环境一样，是为了一个之前的语句能把一个更新后的上下文传给后面。我们还需要处理类似 «if(b){x=1}else{y=2}» 的情况，这里不同的执行路径会产生不同的类型上下文，还有像 «if(b){x=1}else{x=true}» 这

种情况，这里不同的上下文之间会彼此冲突。[5]

根本上说，一个类型系统的限制性和我们能在其中写的程序的表达力之间存在矛盾。一个限制性类型系统可能是好的，因为它保证排除了大量可能的错误，但当它阻止我们写想要写的程序时它又是坏的。一个好的类型系统会在限制性和表达力之间找到可接受的妥协方式，在保持让程序员容易理解的同时排除足够的问题是值得的。

通过坚持简单的思想就可以解决这个矛盾：类型上下文由程序自身之外的什么东西提供，而不能被自身的语句修改。这样确实会排除某些类型的程序，而且明确回避了类型上下文从何而来以及如何得来的问题，但它保持了静态语义的简单性并且给出了一个容易遵守的规则。

那么对于赋值语句，我们说表达式的类型应该与被赋值的变量类型一致：

```
class Assign
  def type(context)
    if context[name] == expression.type(context)
      Type::VOID
    end
  end
end
```

对可以决定每个变量的类型并让它保持不变的所有程序，这个规则足够好，也是一个可以忍受的约束。例如，可以检查在第 2 章中实现了静态语义的 While 循环：

```
>> statement =
     While.new(
       LessThan.new(Variable.new(:x), Number.new(5)),
       Assign.new(:x, Add.new(Variable.new(:x), Number.new(3)))
     )
=> «while (x < 5) { x = x + 3 }»
>> statement.type({})
=> nil
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.type({ x: Type::BOOLEAN })
=> nil
```

9.2.2 好处和限制

已经构建的类型系统可以避免基本的错误。通过根据这些静态语句运行一个程序的玩具版本，可以弄清楚在原始的程序中每一个点上可以出现什么类型的值，并检查这些类型与我们运行它的动态语义将要尝试做的是否正确匹配。这个玩具版本解释的简单性意味着我们只能得到程序求值时可能发生的事情的有限信息，但它还意味着我们很容易进行检查。例如，可以检查一个永远运行的程序：

注 5：一个简单的解决办法是：让类型系统只在它的执行路径产生同样上下文的时候才接受语句。

```
>> statement =
     Sequence.new(
       Assign.new(:x, Number.new(0)),
       While.new(
         Boolean.new(true),
         Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
       )
     )
=> «x = 0; while (true) { x = x + 1 }»
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.evaluate({})
SystemStackError: stack level too deep
```

这个程序确实很傻，但它没包含任何类型错误：循环条件是一个布尔值，并且变量 x 也一直用来存储一个数。当然，类型系统不够聪明，没能告诉我们一个程序是否在做我们想要它干的事情，甚至是否在做有用的事情，而只告诉我们它的各个组成部分是否以正确的方式匹配了。但因为它需要是安全的（就像 Sign 抽象一样），所以有时候对一个程序是否含有任何错误会给出过于悲观的答案。如果用额外的一个语句扩展上面的程序，我们就能看出这一点来：

```
>> statement = Sequence.new(statement, Assign.new(:x, Boolean.new(true)))
=> «x = 0; while (true) { x = x + 1 }; x = true»
>> statement.type({ x: Type::NUMBER })
=> nil
```

方法 #type 返回 nil 表明有错误，因为存在一个把布尔值赋给 x 的语句，可是这个语句永远不会执行，所以在运行时不会实际引发一个问题。我们的类型系统没有那么聪明，认识不到这一点，但它给出了一个安全的答案："这个程序可能会出错。"这过于小心但并没有错误。有时候在程序中试图把一个布尔值赋给一个数字变量确实有可能出错，但因为某种原因，它实际上不会出错。

并不仅仅是无限循环会引起问题。像下面这个程序的动态语义就没有问题：

```
>> statement =
     Sequence.new(
       If.new(
         Variable.new(:b),
         Assign.new(:x, Number.new(6)),
         Assign.new(:x, Boolean.new(true))
       ),
       Sequence.new(
         If.new(
           Variable.new(:b),
           Assign.new(:y, Variable.new(:x)),
           Assign.new(:y, Number.new(1))
         ),
         Assign.new(:z, Add.new(Variable.new(:y), Number.new(1)))
       )
```

```
      )
=> «if (b) { x = 6 } else { x = true }; if (b) { y = x } else { y = 1 }; z = y + 1»
>> statement.evaluate({ b: Boolean.new(true) })
=> {:b=>«true», :x=>«6», :y=>«6», :z=>«7»}
>> statement.evaluate({ b: Boolean.new(false) })
=> {:b=>«false», :x=>«true», :y=>«1», :z=>«2»}
```

变量 x 根据 b 是 true 或者 false 决定来存储一个数字还是一个布尔值，这在求值过程中从来都不是问题。因为程序会一致地使用一个或者另一个；没有可能的执行路径会让 x 既被处理成一个数又被处理成一个布尔值。但静态语义使用的抽象值没有足够的细节，不能展示出这样是可以的[6]，因此安全的近似总是会说"这个程序可能会出错"：

```
>> statement.type({})
=> nil
>> context = { b: Type::BOOLEAN, y: Type::NUMBER, z: Type::NUMBER }
=> {:b=>#<Type boolean>, :y=>#<Type number>, :z=>#<Type number>}
>> statement.type(context)
=> nil
>> statement.type(context.merge({ x: Type::NUMBER }))
=> nil
>> statement.type(context.merge({ x: Type::BOOLEAN }))
=> nil
```



这是一个*静态类型系统*（static type system），为了在运行前就对程序进行检查而设计；在一个静态类型语言中，每一个变量都有相关的类型。Ruby 的*动态类型系统*（dynamic type system）工作方式不同：变量没有类型，而值的类型只是在程序执行过程中它们实际使用时才会检查。这让 Ruby 可以处理赋值给同一变量的不同类型的值，代价就是在程序执行前不能检查出类型的 bug 来。

这个系统专注于编程中某种特定方式的错误：每一段语法的动态语义对其将要处理的值的类型是有某种期望的，而类型系统检查那些期望，以便保证在期望为布尔值的时候不要出现数字，反过来期望为数字的时候不要出现布尔值。但一个程序还存在其他的犯错方式，而这个静态语义并不对其进行检查。例如，这个类型系统不会注意到一个变量在使用之前是否已经被实际赋值了，因此任何包含未初始化变量的程序都能通过这个类型检查器，但在求值过程中则会失败。

```
>> statement = Assign.new(:x, Add.new(Variable.new(:x), Number.new(1)))
=> «x = x + 1»
>> statement.type({ x: Type::NUMBER })
=> #<Type void>
>> statement.evaluate({})
NoMethodError: undefined method `value' for nil:NilClass
```

注 6：在这种情况下，细节是 x 的类型依赖于 b 的值。我们的类型不含有关于变量具体值的任何信息，从而它们无法表达类型和值的依赖。

我们从类型系统得到的任何信息都有些可疑，并且在决定对其投入多大的信任时得注意它的限制。程序静态语义的一次成功执行并不意味着“这个程序将肯定起作用”，只是表明“这个程序在一种特定的方式下肯定不会报错”。能有一个自动化的系统告诉我们程序没有潜在的 bug 或者错误当然很好，但就像在第 8 章看到的那样，世界就是没那么方便。

9.3 应用

本章已经概括了抽象解释的基本思想：使用代价低的近似来了解代价高的计算，并展示了一个简单类型系统作为例子说明近似对分析程序是很有用的。

我们对抽象解释的讨论非常不正式。正式来讲，抽象解释是一种数学化的技术，同样语言的不同语义通过函数连接到一起，这些函数把具体值的集合转换成抽象值的集合，反之亦然。这就允许抽象程序的结果和性质可以按照具体程序的方式来理解。

这项技术一个著名的工业级应用是 Astrée 静态分析器（http://www.astree.ens/fr/），它使用抽象解释自动证明一个 C 程序没有像被零除、数组越界和整数溢出这样的运行时错误。Astrée 不仅已经用来验证为国际空间站运送补给的*儒勒·凡尔纳*（Jules Verne）ATV-001 任务的自动对接软件，还被用来验证空客 A340 和 A380 飞机的飞行控制软件。抽象解释通过提供安全的近似而不是有保证的答案来遵循 Rice 理论，因此 Astrée 有可能报告实际不存在的运行时错误（*错误警告*）；实际上，它的抽象在验证 A340 软件时准确到足以避免任何错误的警告。

用 Simple 语言写的程序只能操纵基本的值（数字和布尔值），因此本章的类型都很基本。现实中的编程语言会处理很多种值，因此真实的静态类型系统要更复杂。例如，像 ML 和 Haskell 这样的静态类型函数式编程语言中函数也是值（就像 Ruby 的 proc），因此它们的类型系统支持*函数类型*。意思就像“带有两个数字参数并返回一个布尔值的函数”，可以让类型检查器校验到一个函数调用中用到的参数与函数定义的参数匹配。

类型系统还可以携带其他信息：Java 有一个*类型与影响系统*（type and effect system）不只跟踪方法参数和返回值的类型，还会跟踪能由方法体抛出的*受检异常*（checked exception，抛出一个异常是一个*影响*），用来保证所有可能的异常要么被处理掉要么被传播出去。

后记

这是我们计算理论之旅的终点了。我们设计了不同能力的语言和机器，从不同寻常的系统中梳理出计算，然后一头扎到计算机编程的理论限制当中。

除了探索特定的机器和技术之外，我们还看到了一些更通用的思想。

- 任何人都可以设计和实现一种编程语言。语法和语义的基本思想是简单的，Treetop 这样的工具可以处理枯燥的细节。
- 每一个计算机程序都是一个数学对象。按句法来说，一个程序只是一个大数；语义上来说，它可能代表一个数学函数，或者一个能被形式化规约规则操纵的分层结构。这意味着数学上的许多技术和成果，如 Kleene 规约理论或者 Gödel 不完备定理，都能等价地应用到程序上。
- 计算，最初被描述为只是“一台计算机做的事”，已经被证明是某种自然力量。很容易把计算想象为一个复杂的人类发明，它只能由对许多复杂部分进行特殊设计的系统来执行，但在系统中还可以看到支持它没那么复杂。因此，计算不是一个枯燥的只是发生在微处理器中的人工过程，而是一个在许多不同地点以不同方式发生的普遍现象。
- 计算不是全有或全无的。不同的机器拥有不同的计算能力，这给了我们用途上的连续性：DFA 和 NFA 有有限的能力，DPDA 更强大，NPDA 还更强大，而图灵机是我们知道的最强大的机器。
- 抽象的编码和级别对于利用计算能力必不可少。计算机是维护抽象宝塔的机器，从非常低层次的半导体物理学开始，上升到层次高得多的多点触控图形用户界面。为了让计算有用，我们需要能把现实世界中复杂的思想编码成机器能处理的更简单的形式，然后再把结果解码回有意义的高层表示。

- 计算能做的事情是有限制的。我们不知道如何构建比图灵机能力更强的机器，但确实存在图灵机无法解决的问题，而这些问题包括发现我们所写程序的信息。可以利用模糊的或者不完整的答案处理这些限制，以便质疑我们程序的行为。

这些思想可能不会立即改变你工作的方式，但我希望它们已经满足了你的某种好奇心，并且能帮助你享受在宇宙中实现计算时所度过的时光。

Computers/Programming Languages

计算的本质 深入剖析程序和计算机

我知道你是一位编程高手，写代码对你而言是手到擒来的事。但是，你确定自己多年练就的编程技能不是建立在某种想当然的假设基础上？确定自己不是每天都在“稀里糊涂”地写代码？确定真正理解自己的代码是如何运行的吗？

如果你想像“大牛”级的程序员一样做开发，或者想摆脱自己半路出家的知识“囧”境，本书能够为你真正讲明白计算理论和编程语言的工作原理与真切含义。本书使用简单的Ruby代码做示例，没有枯燥难记的数学符号。作者极力推崇循序渐进和从实践中学习，他从机器、语言讲到程序，又一路从最简单的机器（有限自动机）过渡到复杂的机器（图灵机），从设计实现简单的编程语言到极简的机器，而后又推理所谓“不可能”解决的问题，为读者完美打造了轻松有趣的阅读体验。

- 计算的基本概念，如语言中的图灵完备性
- 程序如何使用动态语义与机器交流思想
- 有限自动机的功能
- 通用图灵机如何催生了今天的通用计算机
- 使用简单语言和细胞自动机执行复杂计算
- 哪些语言特性对计算而言是必不可少的
- 停机和自引用为何会让计算问题无解
- 用抽象解释和类型系统分析程序

Tom Stuart

伦敦数字产品咨询公司Codon的创始人、计算机科学家、程序员，擅长Ruby、Rails、Web应用、用户体验、面向对象设计和行为驱动开发。另外，作为顾问、导师和培训师，他经常通过网络帮助各家公司高质高效地创建软件产品。他还曾在剑桥大学做编译器优化方面的演讲，与人联合组织过Ruby大会（Ruby Manor），而且是伦敦Ruby用户组的成员。

封面设计：Randy Comer　马冬燕

O'REILLY®
oreilly.com.cn

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议　计算机/计算机科学/计算理论

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-36154-7



ISBN 978-7-115-36154-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks